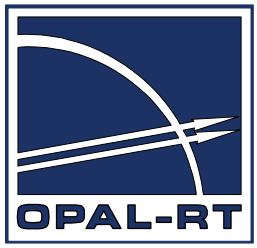
RT-LAB

7.0

User's Manual



WWW.OPAL-RT.COM SUPPORT@OPAL-RT.COM 1-877-935-2323 (08:30-17:30 EST) 1-514-935-2323

Quick Start

The *MetaController* application must be running when using RT-LAB software. Start this application from the RT-LAB program group.

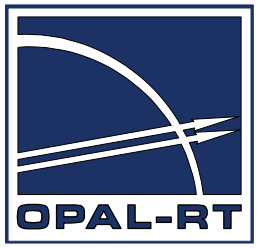
To use RT-LAB:

1. Open your Simulink or SystemBuild block diagram model.
2. Group the model into one Console, one Master, and one or more Slave subsystems. Name each subsystem (**`SC_`**, **`SM_`**, **`SS_`**) and save the top-level grouped model.
3. Start the Main Control Panel by double-clicking on the Desktop's *RT-LAB MainControl* program icon.
4. Click on the filename to select your grouped model file, and click *Open Model*.
5. Click on the *Compile* button in the Main Control dialog box. Set parameters as required. The message `Compilation finished...` will appear on the screen when the model has successfully been compiled.
6. Click on the *Assign Nodes* button to assign each computation subsystem to a physical node. A dialog box will appear to allow you to assign subsystems to nodes: double-click the physical node you want to use from the list of *Remaining Physical Nodes*.
7. In the Main Control Dialog box, make sure the *Use model Console* option is selected. Click on the *Load* button to load the files to be executed by the various nodes.
8. Click on the *Execute* button in the Main Control Dialog box to run your simulation. The message displayed in the RT-LAB display window should confirm that the model is in `Run` mode.
9. Click on the *Play* button in the Console if it is not started.
10. To stop execution of the model's simulation, click on the *Reset* button in the Main Control Dialog box. To quit the program once a model has been reset, click on the *Close* button in the Main Control Dialog box. To open another model without closing RT-LAB, click on the *Disconnect* button in the Main Control Dialog box, to disconnect the first model and then open the next model.

RT-LAB

7.0

User's Manual



WWW.OPAL-RT.COM SUPPORT@OPAL-RT.COM 1-877-935-2323 (08:30-17:30 EST) 1-514-935-2323

Publication Information

© Opal-RT Technologies Inc., 2002. All rights reserved.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. This documentation contains proprietary information of Opal-RT Technologies Inc. No part of this documentation may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, recording or otherwise, without prior written permission of Opal-RT.

The information in this document was prepared by Opal-RT with all reasonable care, and is believed to be accurate. However, Opal-RT does not assume responsibility for losses or damages resulting from any omissions, inaccuracies, or errors contained herein. This documentation is subject to change, and revisions or new editions may be issued to incorporate such changes.

Opal-RT and RT-LAB are trademarks of Opal-RT Technologies Inc. All other product or brand names are trademarks or registered trademarks of their respective holders.

Printed in Canada.

Lead Software Architect:	François Desruisseaux
Documentation by:	Loïc Schoen, Gaspard Turmel, Alexandre Abreu, Julien Darrah, Mathieu Dubé-Dallaire, Mathieu Fortin, Pierre Grondin, Fawzi Hassaine, Jason Michel Lambert, Nicolas Léchevin, Djibril Ndiaye, Michel Ouellet, Irène Pérès, Francis Provencher, Biao Yu, Adam Zembala, Paul Baracos
Layout:	Janelle Mellamphy
Title:	RT-LAB v7.0 User's Manual
Publication ID:	DO-020108JM-1c
Date:	April 2004
Revision:	A

Table of Contents

Quick Start Q

Table of Contents iii

List of Figures vii

List of Tables ix

1 Introduction 1

1.1 The Evolution of Real-Time Simulation 1

1.2 How to Use the User's Manual 2

Notation Conventions 3

1.3 Getting More Help 3

2 Fundamentals 4

2.1 How RT-LAB Works 4

The Command Station 4

Target nodes 4

Compilation Node 5

Communication 5

FireWire real-time link 5

Ethernet link 5

I/O boards 5

2.2 How RT-LAB is Used 6

Designing and validating the model 6

Using block diagrams 6

Using I/O devices 6

Running Simulations 7

RT-LAB simulation 7

Using the Console as a graphic interface 7

2.3 Required Knowledge 7

Windows NT/2000/XP 7

Simulink or SystemBuild 8

MATLAB or MATRIXx 8

I/O modules 8

3 **Building a Distributed Model for RT-LAB** **10**

3.1 Minimum Step Size Calculations . . . 11
How to calculate approximate step size . . . 11
Minimum step size . . . 11
Hardware Synchronization . . . 12
3.2 Grouping the Model into Subsystems . . . 13
To keep in mind when dividing your model . . . 14
3.3 OpComm Communication Blocks . . . 15
Inserting OpComm blocks . . . 16
Rules for inserting OpComm blocks . . . 17
OpComm parameters for Simulink users . . . 17
OpComm parameters for SystemBuild users . . . 18
Integer parameters for SystemBuild OpComm blocks . . . 19
SystemBuild UCBs . . . 19
3.4 Maximizing Parallel Communication . . . 20
Input-dependent data . . . 20
State variables . . . 22
Delay blocks . . . 23
Key points about network communication . . . 23
Prioritizing data transmission with Simulink . . . 23
Prioritized data transmission in SystemBuild . . . 24

4 **RT-LAB Blocks** **25**

4.1 I/O and Synchronization Module blocks . . . 25
Using I/O blocks in RT-LAB . . . 26
I/O block parameters by card type . . . 26
IP Modules . . . 26
ISA card . . . 27
PCI card . . . 27
PC/104/104 card . . . 27
I/O Synchronize option . . . 27
4.2 RT-LAB I/O and Synchronization Module blocks . . . 28
OpAnalogIn . . . 28
OpAnalogOut . . . 29
OpDigitalIn . . . 30
OpDigitalOut . . . 31
OpQuad-Decoder . . . 31
OpSync_VME . . . 32

5 **Acquiring and Viewing Data** **33**

5.1 Acquisition Groups . . . 33
Acquisition parameters . . . 33

Triggering Acquisition . . . 33
Recording data with OpWriteFile . . . 34
5.2 Understanding Data Reception . . . 34
Communication between the Console and the Target nodes . . . 34
Synchronous Acquisition . . . 37

6 **Models with Multiple Sampling Rates** **40**

6.1 Requirements for running models in Multithread, Single CPU mode . . . 40
Determining base sampling rate . . . 41
Making proper rate transitions . . . 41
Compiling and Loading the model . . . 41
6.2 Requirements for running models in Single-Thread, Multi-CPU mode . . . 42
Inserting blocks for proper rate transitions . . . 42
Communication rate . . . 42
6.3 Requirements for running models in Multithread, Multi-CPU mode . . . 44
6.4 Multirate mode options . . . 45
Using an SC_ Subsystem in Multirate Mode . . . 45
Using I/O blocks in Multirate Mode . . . 45

7 **Running Your Simulation** **47**

7.1 Compiling . . . 47
7.2 Simulating Offline . . . 48
Loading your model and executing its simulation . . . 48
Interacting with the model during execution . . . 49

8 **Sample Simulation** **51**

A1 **Setting up a Serial Connection to an RT-LAB Target** **57**

A2 **Coding an RT-LAB icon to Make I/O and Timer Icons** **58**

A2.1 Developing icons with RT-LAB . . . 58
Required Knowledge . . . 59
A2.2 Simulink S-Functions . . . 59
Main S-Function Routines . . . 59
mdlInitializeSizes . . . 60
mdlInitializeSampleTimes . . . 60
mdlInitializeConditions . . . 60
mdlOutputs . . . 60

mdlTerminate 60
How to create an S-Function 61
A2.3 SystemBuild UCBs 61
A2.4 Creating icons with RT-LAB 62
RT-LAB Routines 62
Register_CB_Init 62
Register_CB_Close 62
Register_CB_ExecuteIO 63
Register_CB_PauseIO 63
A2.5 Programming an RT-LAB S-Function for an I/O icon 63
Template Header 64
InitializeConditions 65
MdlOutputs 66
mdl_terminate 66
MyExecuteIO 67
MyFunctionPause 67
A2.6 Programming an RT-LAB S-Function for a Timer icon 68
Template Header 68
InitializeConditions 70
mdl_outputs 72
mdl_Terminate 72
WaitForTimer 73
DisableTimer 74

Glossary **75**

Index **81**

List of Figures

Figure 3-1 Some Simulink blocks that may be included in the Console subsystem. 13
Figure 3-2 Block model divided into one Console, one Master, and two Slave subsystems.. 14
Figure 3-3 Inside "view" of the Console, Master, and Slave. 16
Figure 3-4 SystemBuild OpComm block parameters. 18
Figure 3-5 This model simulates sequentially since its data are all input-dependant.. 20
Figure 3-6 Event chart for the sequentially simulated model in Figure 3-5.. 21
Figure 3-7 Distributed model, including state variables identified to be sent as a priority. 22
Figure 3-8 Event chart for the distributed model shown in Figure 3-7. 22
Figure 3-9 Output Out1 shown on the left is an input-dependent output; the _s added to the Out1 shown on the right declares this output as a priority send. 24
Figure 4-1 OpAnalogIn block. 28
Figure 4-2 OpAnalogOut block. The module in this figure would maintain its last output value when paused and reset, since their inputs have the same value as the *Run* input. 29
Figure 4-3 OpDigitalIn block.. 30
Figure 4-4 OpDigitalOut block. OpDigitalOut maintains its last output value with *Pause* and *Reset*. 31
Figure 4-5 OpQuad-Decoder block. 32
Figure 4-6 OpSync_VME block. 32
Figure 5-1 Expected signal in the data display. 35
Figure 5-2 A "hole" is generated in the data display. 35
Figure 5-3 Increasing the effective length of a frame by increasing the decimation factor. 36
Figure 5-4 Display resolution error that can result from increase in decimation factor. 36
Figure 5-5 Signal with no aliasing. 37
Figure 5-6 Sine waves showing aliasing due to a non-synchronized time axis. . . . 38
Figure 5-7 Interpolation in data acquisition. 39
Figure 6-1 Specifying a Fixed step of 0.001 with Multitasking. 41
Figure 6-2 The Master runs at 1ms and the Slaves run at 10ms in this model. . . . 43
Figure 6-3 Multithread, Multi-CPU: possible sampling times in a model that would run on 3 CPUs. 44
Figure 6-4 Rate transition for sending data to the Console subsystem. 45
Figure 6-5 I/O in Multitasking mode. 46
Figure 6-6 I/O executing at the base rate of the Master. 46
Figure 7-1 Slider gain. 49
Figure 7-2 Modifying feedback gains using the Console.. 50
Figure 8-1 Grouping of an original model.. 52

Figure 8-2 Final regrouped model. 52
Figure 8-3 rt_demo1 scope display. 55
Figure A2-1 Execution of a Simulink model 60

List of Tables

Table 3-1	SystemBuild OpComm integer parameters	19
Table 4-1	OpAnalogIn I/Os	28
Table 4-2	OpAnalogOut I/Os	29
Table 4-3	OpDigitalIn I/Os	30
Table 4-4	OpDigitalOut I/Os	31
Table 4-5	OpQuad-Decoder I/Os	32
Table 4-6	OpSync_VME I/Os	32
Table 5-1	OpComm icon mask options	38

1 Introduction

1.1 The Evolution of Real-Time Simulation

In a competitive world, accurate prediction provides a significant advantage. Simulation technology can cut costs, minimize risks, improve reliability, and bring a system into operation more quickly.

Providing insight into the design of processes, architectures, or product lines before significant time and cost have been invested, simulation systems make it possible to address issues before they become problems, and before investing large sums for implementation.

The development of analog computers during the mid-twentieth century was a defining moment in the progress of simulations technology. With the emergence of digital computers in the 1950s, simulation systems were increasingly refined, and the progress has continued to expand. Just over a decade ago, simulations were developed through textual coding of the model; since the early 1990s, graphic simulation tools have become the norm.

Opal-RT furthers simulation technology progress, designing flexible and affordable tools to make possible real-time simulations. Our methodology is guided by the following principles:

- The key to real-time simulation success will be advanced modeling and simulation algorithms, not hardware.
- Real-time simulation and control should be easy and affordable.
- Simulation architecture should be scalable.
- Simulation tools should look after low-level details and code automatically, taking users from graphic models to hardware without requiring coding by hand.
- Control system models should be deployable in real life as embedded control systems running on industrial-grade real-time platforms.
- Software should be flexible enough to allow users to benefit from advances in commercial-off-the-shelf components.
- In real-time, every microsecond counts.

From energy production, to aerospace and aeronautics, to robotics, automotive, naval, and defense: real-time simulation and support for hardware-in-the-loop modeling are increasingly recognized as essential tools for design in these and other industries.

Opal-RT has developed the technology that enables model separation to allow distributed execution, while automatically generating, downloading, and running real-time, high-performance, deadlock-free distributed simulation software code. Our software integrates the most advanced computer, communication, and simulation technologies commercially available, to reduce design and implementation time and cost, while increasing scalability and flexibility – with no sacrifice in performance.

1.2 How to Use the User's Manual

The first chapters of this manual provide the information required to get started:

- Chapter 2, Fundamentals gives an overview of RT-LAB;

The next two chapters explain how to prepare your model for simulation with RT-LAB:

- Chapter 3, Building a Distributed Model for RT-LAB takes you from your Simulink or SystemBuild block diagram to a distributed model ready for RT-LAB; and
- Chapter 4, RT-LAB Blocks lists the I/O, synchronization blocks, and devices available with RT-LAB, and explains how to use.

Chapter 5, Acquiring and Viewing Data, explains concepts required to refine your simulation, and provides RT-LAB solutions to potential data reception problems.

Working with multi-rate models is clarified for Simulink users in Chapter 6, Models with Multiple Sampling Rates, and Chapter 7, Running Your Simulation takes you through the final preparations, from compiling the model to running its simulation. Finally, to put all the above information into practice, Chapter 8, Sample Simulation walks users through the simulation process from beginning to end.

A Glossary of the terms used within this documentation is provided at the end of this manual.

For experienced users who wish to get RT-LAB running quickly, Quick Start instructions can be found on the reverse of this manual's cover page.

1.2.1 Notation Conventions

Style conventions as used in this manual are shown below, with examples of their use:

Convention:	**Example:**
Text a user reads on-screen appears in Courier.	Wait for the message `Done`.
Text a user enters into a field appears in Courier bold.	Type **`exit`** to close.
Menus and menu items appear in Franklin Gothic.	Select File > Print.
Buttons and fields appear in Franklin Gothic italics.	Click *Open*.
Keyboard keys appear enclosed in angle brackets.	Press <Enter>.
Filenames and folders appear in Univers.	Open the file model.sbd.
Reference sources appear in Times italics.	See *Chapter*.
Notes are identified by a lightbulb icon.	
Important troubleshooting hints are identified by a lightbulb-with-exclamation-mark icon.	

1.3 Getting More Help

If you need more information, contact Opal-RT Customer Support:

Toll-Free (U.S. and Canada)	1-877-935-2323 (08:30-17:30 EST)
Phone	1-514-935-2323
Fax	1-514-935-4994
Email	sales@opal-rt.com info@opal-rt.com support@opal-rt.com
Mail	1751 Richardson, suite 2525 Montréal (QC) Canada H3K 1G6
Web	www.opal-rt.com

2 Fundamentals

This chapter describes the fundamentals of RT-LAB, providing an overview of the simulation process, from designing and validating the model, to using block diagrams and I/O devices, to running the simulation and using the Console as a graphic interface. Related hardware functions are also discussed, including the Command Station and target nodes, communication links, and I/O boards. Finally, the knowledge required about select hardware and software components is outlined at the end of this chapter.

2.1 How RT-LAB Works

RT-LAB software runs on a hardware configuration consisting of Command Station, compilation node, target nodes, the communication links (real-time and Ethernet), and the I/O boards.

RT-LAB software is configured on a Windows NT or Windows 2000 computer (Windows 2000 is an NT platform, so when this Manual refers to Windows NT, it is referring to either Windows NT or Windows 2000) called the Command Station. Simulations can be run entirely on the Command Station computer, but they are typically run on one or more target nodes. For real-time simulation, the preferred operating system for the target nodes is QNX. When there are multiple QNX nodes, one of them is designated as the compilation node. The Command Station and target node(s) communicate with each other using communication links, and for hardware-in-the-loop simulations target nodes may also communicate with other devices through I/O boards.

2.1.1 The Command Station

The Command Station is a PC workstation that operates under Windows NT, and serves as the user interface. The Command Station allows users to:

- edit and modify models;
- see model data;
- run the original model under its simulation software (Simulink, SystemBuild, etc.);
- generate code;
- separate code;
- control the simulator's *Go/Stop* sequences.

2.1.2 Target nodes

The target nodes are real-time processing and communication computers that use commercial processors interconnected by an Ethernet adapter. These computers can also

include a real-time communication interface like FireWire or cLAN (depending on the selected OS), as well as I/O boards for accessing external equipment. The real-time target nodes perform:

- real-time execution of the model's simulation;
- real-time communication between the nodes and I/Os;
- initialization of the I/O systems;
- acquisition of the model's internal variables and external outputs through I/O modules;
- implementation of user-performed online parameters modification;
- recording data on local hard drive, if desired;
- supervision of execution of the model's simulation, and communication with other nodes.

2.1.2.1 Compilation Node

The compilation node is used to:

- compile C code;
- load the code onto each target node;
- debug the user's source code (S-function, User Code Block, etc.).

2.1.3 Communication

2.1.3.1 FireWire real-time link

The real-time communication link works using FireWire (IEEE P-1394) or cLAN interfaces. Both ensure:

- real-time communication between slaves, and between the target nodes and the I/O;
- synchronization between the I/O boards and target nodes.

2.1.3.2 Ethernet link

An Ethernet link is used to transfer simulation models and run-time data between the Command Station and the target nodes.

2.1.3.3 I/O boards

Both analog and digital I/O boards are supported by RT-LAB. These allow connection to external equipment for applications such as HIL. See 4, RT-LAB Blocks for a detailed description of I/O boards, and how to install them.

2.2 How RT-LAB is Used

RT-LAB is an industrial-grade software package for engineers who use mathematical block diagrams for simulation, control, and related applications. The software is layered on top of industry-proven commercial-off-the-shelf (COTS) components like popular diagramming tools MATLAB/Simulink and MATRIXx/SystemBuild, and works with viewers such as LabVIEW and Altia, and programming languages including Visual Basic and C++.

2.2.1 Designing and validating the model

The starting point for any simulation is a mathematical model of the system components that are to be simulated. Users design and validate a model by analyzing the system to be modelled, and implementing the model in the dynamic simulation software. RT-LAB is designed to automate the execution of simulations for models made with offline dynamic simulation software, like Simulink or SystemBuild, in a real-time multiprocessing environment. RT-LAB is fully scalable, allowing users to separate mathematical models into blocks to be run in parallel on a cluster of machines, without subtly changing the model's behavior, introducing real-time glitches, or causing deadlocks.

2.2.2 Using block diagrams

Using block diagrams for programming simplifies the entry of parameters, and guarantees complete and exact documentation of the system being modeled. Once the model is validated, the user separates it into subsystems and inserts appropriate communication blocks. Each subsystem will be executed by target nodes in RT-LAB's distributed system.

2.2.3 Using I/O devices

RT-LAB supports the use of Input/Output devices to enable the integration of external physical components into the system. This arrangement is commonly known as a Hardware-in-the-Loop (HIL) configuration. Real-time simulations that use HIL reduce the need to simulate processing components, since part of the simulation is actual hardware. The optimized use of I/O devices allows RT-LAB to work as a programmable control system that presents a flexible real-time user-machine interface.

Interfaces for I/O devices are configured through custom blocks that need only be added and connected to the graphic model's blocks. RT-LAB's automatic code generator will map the model's data onto the physical I/O cards.

Note: For a complete discussion of I/O devices, see 4, RT-LAB Blocks.

2.2.4 Running Simulations

Once the original model has been separated into subsystems associated with the various processors, each portion of the model is automatically coded in C, and compiled for execution by the target nodes.

Target nodes are commercial PCs, equipped with PC-compatible processors, that operate under a Windows NT or a QNX environment. In the QNX environment, the real-time sending and reception of data between QNX nodes is performed through FireWire-type communication boards, typically at 200 Mb/s or 400 Mb/s (depending on the card chosen). In a Windows NT environment, real-time communication takes places using a Giganet communication device (1,200 Mb/s), or shared memory for distributed computing on a single multiprocessor machine. Less time-critical data can be sent and received through Ethernet communication.

For more information about running your simulation, see 7, Running Your Simulation.

2.2.4.1 RT-LAB simulation

When the C coding and compilation are complete, RT-LAB automatically distributes its calculations among the target nodes, and provides an interface so users can execute the simulation and manipulate the model's parameters. The result is high-performance simulation that can run in parallel and in real-time.

2.2.4.2 Using the Console as a graphic interface

Users can interact with RT-LAB during a simulation by using the Console, a command terminal operating under Windows NT. Communication between the Console and the target nodes is performed through a TCP/IP connection. This allows users to save any signal from the model, for viewing or for offline analysis. It is also possible to use the Console to modify the model's parameters while the simulation is running. (For complete details, see 5, Acquiring and Viewing Data.)

2.3 Required Knowledge

RT-LAB is designed to integrate numerous peripheral COTS components. For the best possible use of RT-LAB, basic understanding of these hardware and software components is required.

2.3.1 Windows NT/2000/XP

Since RT-LAB works under the Windows NT/2000/XP operating system, the user must be familiar with this environment and its tasks, including file access and execution, process management through Task Manager, etc. Knowledge of TCP/IP network communication is an added advantage.

2.3.2 Simulink or SystemBuild

Simulink and SystemBuild are software packages that enable modeling, simulation, and analysis of dynamic systems. Users describe their models graphically, following a precise format based on a library of blocks. RT-LAB uses Simulink or SystemBuild to define models that will be executed by the real-time multiprocessing system, and defines its own simulation parameters through Simulink's or SystemBuild's. It is expected that users have a clear understanding of Simulink's or SystemBuild's operation, particularly as regards model definition and the model's various simulation parameters. Users who intend to create their own icons to interact with RT-LAB should know how to create Simulink or SystemBuild icons (S-functions and UCBs) for both the Command Station and target environments.

2.3.3 MATLAB or MATRIXx

MATLAB and MATRIXx are technical computing software packages that integrate programming, calculation, and visualization. MATLAB and MATRIXx also include Simulink and SystemBuild, respectively; these software packages are discussed above. As RT-LAB works in conjunction with these environments to define models, the user must be familiar with aspects of MATLAB as related to Simulink, and aspects of MATRIXx as related to SystemBuild. Users intending to gather data from one system and then process the data offline, for example, must know how to save the data and retrieve it using MATLAB or MATRIXx, and to display data through the different on-screen tools available within the software.

2.3.4 I/O modules

Those intending to use RT-LAB in a HIL context must know how to specify the characteristics of their communication boards, and those of the attached modules. Since I/O communication is defined through special blocks in the model, characteristics such as the function and location of each IP module are specified as parameters for these blocks.

Note: RT-LAB works with many different COTS I/O products (standard I/O boards, carrier boards, com cards, etc.). Vendor product sheets and links to vendors' sites are available on the Opal-RT web site, at www.opal-rt.com/p_rt-lab_devices.html

When the model is separated in RT-LAB, each subsystem is assigned to one machine (or target node) within a cluster. That is, once the user has separated the model into subsystems, RT-LAB asks to which target node each subsystem should be downloaded. Elements are distributed among the subsystems according to their type:

- all display elements are placed within the one Console (SC_) subsystem;
- high-rate elements are placed within the one Master (SM_) subsystem; and
- all other elements are distributed into the Master and/or one or more Slave (SS_) subsystems.

The number of nodes in the cluster is restricted only by the number of processors available and any limit imposed by the communication links. For complete details on how to distribute your model, see 3, Building a Distributed Model for RT-LAB.

3 Building a Distributed Model for RT-LAB

Any Simulink or SystemBuild model can be implemented in RT-LAB, but some modifications must be made in order to distribute the model and transfer it into the simulation environment. The success of distributed computing with a complex model will depend on the separation of that model into small subsystems synchronized to run in parallel. This should be kept in mind early in and throughout the design process.

To build an RT-LAB model, users must modify the block diagram (.mdl or .sbd file) by regrouping the model into calculation subsystems, inserting OpComm communication blocks, taking into account step size calculations, and exchanging state variables or using *Delay* blocks; each of these topics is covered within this chapter. After these steps have been completed, RT-LAB begins the compilation process with the regrouped file, separating the model, and generating and compiling the code. The user then sets execution settings, which are covered in the following chapters, after which point the model's simulation is ready to be executed.

This chapter explains the steps required to build an RT-LAB model by modifying your Simulink or SystemBuild block diagram to ensure it is properly separated for distributed execution, and to maximize its performance:

1. Calculate the required step time.

   Step size must be calculated to optimize communication between nodes and to determine the approximate simulation speed.

2. Group the model into subsystems.

   In this step, you graphically group into subsystems the calculations to be performed by a given CPU.

3. Insert OpComm communication blocks.

   Communication blocks are part of a block library specifically defined for the RT-LAB interface. They are used by RT-LAB to identify parameters required for communication between the nodes in the hardware configuration.

4. Identify required outputs as state variables.

   In order to allow for real-time synchronization, and to maximize the calculation speed of the system, all data exchanged between the different nodes of the real-time network must be in the form of state variables. State variables are variables that do not depend on inputs for a given calculation step, meaning that they can be calculated and sent at the very beginning of that given step. (Note that in a Simulink model, if the output from a block functions has a direct feedthrough, it is *not* a natural state variable.)

3.1 Minimum Step Size Calculations

The key behind RT-LAB is distributing a simulation's computations across a cluster of computers. To determine how fast a simulation can run, step size calculations must be done to synchronize flow of data between the network's target nodes.

To balance the workload across your network, each node should be responsible for an equal amount of computation. Step size calculations must be done in order to ensure uninterrupted and synchronized communication between nodes, and to have a general idea of how fast the simulation can run.

Step size determines the time interval between subsystems' consecutive calculation steps. Models can have a variable or fixed step size:

- a model with a variable step size can modify its step size during its simulation;
- a model with a fixed step size will use the same step size value throughout its simulation.

3.1.1 How to calculate approximate step size

The basic equation for calculating step size is:

$$\text{step size} = \frac{\text{computation time } (\mu s)}{\text{number of nodes}} + \text{system overhead}$$

The simulation speed on the real-time system depends on the speed and the number of CPUs available. It usually runs faster than the simulation on Simulink. Assume that the computation time of one step of the model in Simulink is 20000µs (**20ms**), a realistic value for a complex robot controller with trajectory generation. Since the model is complex, assume that the speed gain due to running the model as an executable under a real-time operating system will be about 20. That makes our computation time on the target equal to 20000 / 20 = **1000 µs.** Let us assume for this example that a minimum of data is bring exchanged over the FireWire network, so that we can neglect transmission time.

3.1.1.1 Minimum step size

Minimum step sizes will vary depending on the number of target nodes that make up your network, since complex node synchronization will involve more overhead. *As a general rule*, an additional CPU will add another 20 µs to the minimum step size. Consider the following example:

If, for a single node, the minimum step size = 15 µs

> one node, no FireWire transmission, no synchronization; model running as fast as possible.

then for 2 nodes, the minimum step size = 100 µs

> constants sent from slave node to master node and returned to the Slave node.

and for 3 nodes, the minimum step size = 120 µs

> 2 nodes, slave node sending 1 data (double) to master node, and the Master node is returning this data to the slave node 2.

Note: The results will be slightly better with a real model, however, since Slave nodes would not send their data at the same time.

3.1.2 Hardware Synchronization

When using hardware, make sure that data on the I/O board is acquired with the same step size as the model. A timer card can be used to generate an interrupt message to synchronize all access to I/O boards. Numerous timers are supported; the ISA ACL-8454 (using 8254 timer chip) can be used as a default.

Note: For I/O cards with integrated timers that Opal-RT does not already support, our engineers can custom-write a driver, or you could write your own using our template.

Step size for running the model on 1 node:

1000 µs computation time

\+ 15 µs (overhead for 1 node system)

= **1015 µs**

Step size for running the model on 2 nodes:

1000 µs computation time

/ 2 nodes (assuming each one is waiting exactly the same amount of computation)

= 500 µs

\+ 100 µs (overhead for 2 nodes system)

= **600 µs**

Step size for running the model on 3 nodes:

1000 µs computation time

/ 3 nodes (assuming each one is waiting exactly the same amount of computation)

= 335 µs

\+ 120 µs (overhead for 3 nodes system)

= **455 µs**

3.2 Grouping the Model into Subsystems

The model must be divided into subsystems, each of which represents one node in the real-time network. Three types of subsystems are available:

Console:

- The Console subsystem is the station operating under Windows NT, where the user interacts with the system. It contains all the Simulink/SystemBuild blocks related to acquiring and viewing data (scope, manual switch, To Workspace-type blocks, etc.). Any of these blocks that are required by the user, whether during or after the execution of the real-time model, should be included in the Console subsystem. There can be only one Console per model.

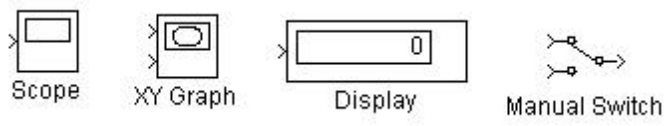


Figure 3-1 Some Simulink blocks that may be included in the Console subsystem.

Master:

- The Master computation subsystem is responsible for the model's real-time calculation and for the overall synchronization of the network. In a system containing Hardware-in-the-Loop (HIL) this subsystem is also responsible for I/O communication. The Master includes Simulink or SystemBuild blocks that represent operations to be performed on signals and/or on I/O icons. There can be only one Master subsystem per model.

Slaves:

- Slave computation subsystems are also responsible for performing calculations in the model, and are driven by the Master subsystem, which synchronizes the whole network. Slave computation subsystems contains Simulink or SystemBuild blocks that represent operations to be performed on signals. Slave subsystems can include I/O blocks, but while these will be synchronized within the Slave subsystem, they will not be synchronized with I/O communication within the Master subsystem. A CPU may contain several Slave subsystems, although this set-up would not allow for distributed processing, and should therefore be used only in simulation mode. For real-time applications, there can be as many Slaves as there are CPUs.

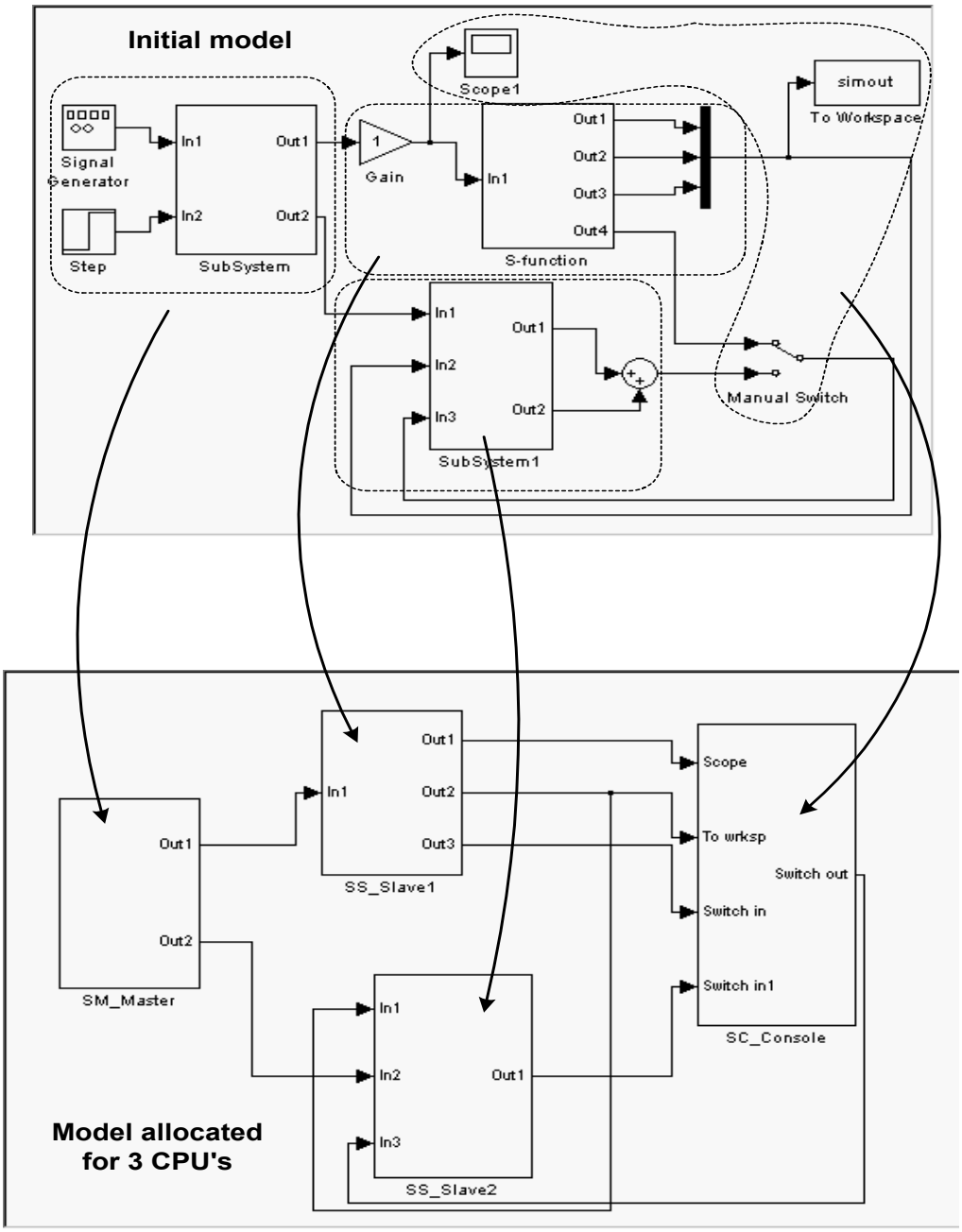


Figure 3-2 Block model divided into one Console, one Master, and two Slave subsystems.

3.2.1 To keep in mind when dividing your model

A number of conventions must be followed when organizing and naming the subsystems. Consider the following example of a simple model divided for distribution on three target nodes:

- All blocks must be included in the subsystems: the top-level model must only show the grouped subsystems. Name the subsystems with a prefix indicating the function of that subsystem: SC_, SM_, or SS_ for Console, Master, and Slave subsystems, respectively.
- With RT-LAB for SystemBuild, the model must contain only one top-level superblock. All the SC_, SM_ and SC_ superblocks should be located one level below this top superblock in the model superblock's hierarchy.
- With RT-LAB for SystemBuild,.
- A model must have one Console subsystem (SC_) and one Master calculation subsystem (SM_). All other subsystems must be Slave calculation subsystems (SS_).
- None of the three types of subsystems (SC_, SM_, or SS_) may include any other type of subsystem. No RT-LAB calculation subsystem (SM_ or SS_) may include another RT-LAB calculation subsystem.

3.3 OpComm Communication Blocks

This section explains how and when to insert OpComm blocks into your block diagram, and discusses OpComm parameters that are specific to Simulink users, and those specific to SystemBuild users.

The following figure demonstrates these points with an example of OpComm icon inserted in a Simulink model.

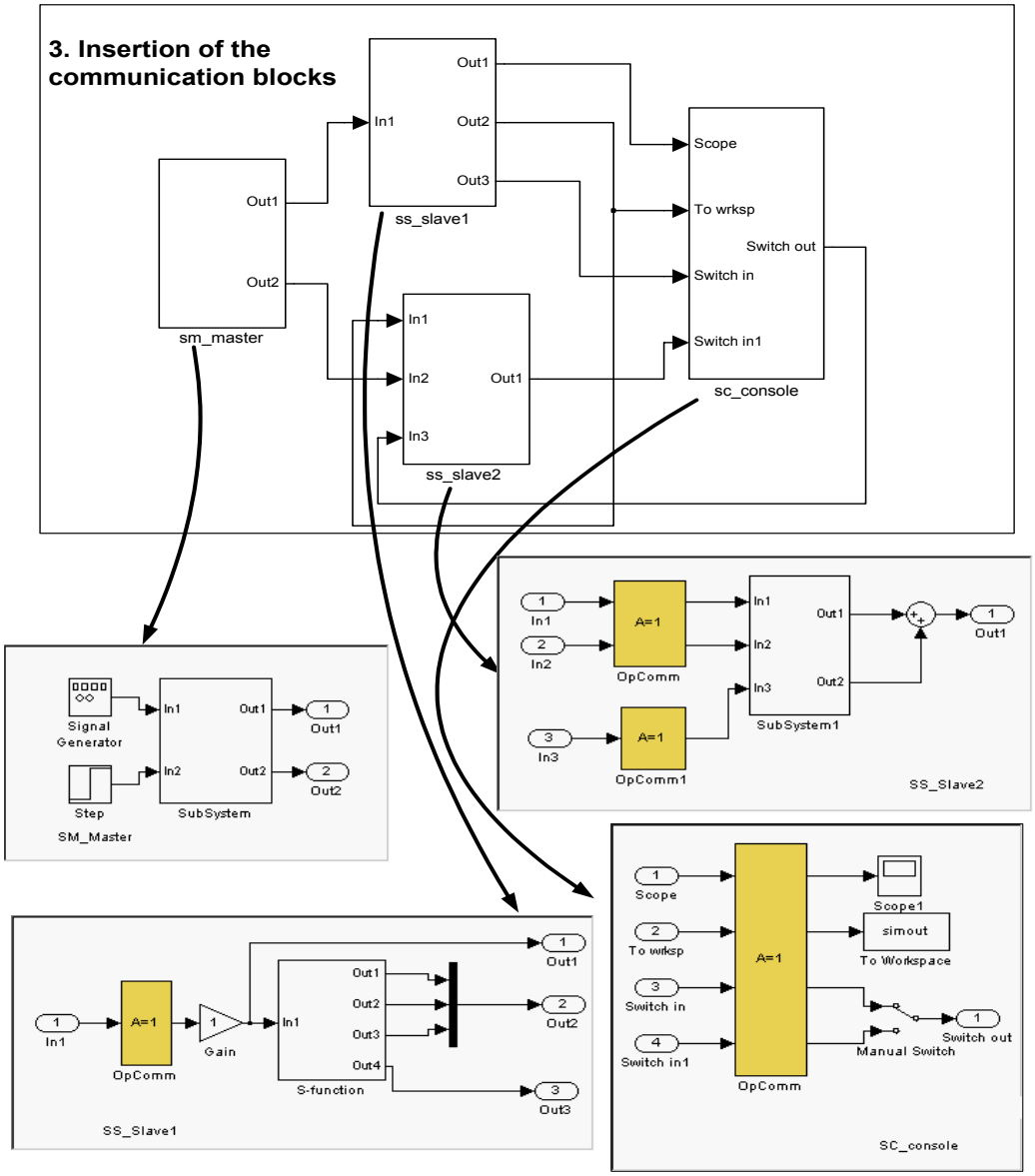


Figure 3-3 Inside "view" of the Console, Master, and Slave.

3.3.1 Inserting OpComm blocks

Once the model is grouped into Console and computation subsystems, special blocks called OpComm blocks must be inserted into the subsystems. These are simple feed-through blocks that intercept all incoming signals before sending them to computation blocks within a given subsystem. OpComm blocks serve three purposes:

1. When a simulation model runs in the RT-LAB environment, all connections between the main subsystems (SC_, SM_, or SS_) are replaced by hardware communication

links. For communication between real-time target nodes (SM_ and SS_) RT-LAB uses a designated real-time link. For communication between the Console (SC_) and the real-time nodes (SM_ or SS_) RT-LAB uses TCP/IP.

Because of these communication links between nodes, the simulation might not run the same way in Simulink or SystemBuild as in RT-LAB. In RT-LAB, a computation subsystem waits for reception of all signals before it is able to start calculating. In Simulink and SystemBuild, on the other hand, computations performed on a signal start as soon as the signal is available. As pass-through icons that wait for all inputs before making their outputs available, OpComm blocks emulate the behavior of the system as it is run in RT-LAB. Keep this effect in mind when designing models.

If you receive a message indicating that the inserted OpComms create an algebraic loop, this means your model will not be able to run in real-time, as a deadlock has been created. To correct this problem, include a state variable. See the full discussion in section 3.4, Maximizing Parallel Communication.

2. OpComm blocks provide information to RT-LAB concerning the type and size of the signals being sent from one subsystem to another.

3. OpComm blocks inserted into the Console subsystem allow you to select the data acquisition group you want to use to acquire data from the model, and to specify acquisition parameters. (Acquisition parameters are described in the following sections, specific to Simulink or SystemBuild, respectively.)

3.3.2 Rules for inserting OpComm blocks

1. When inserting OpComm blocks into the Console (SC_) subsystem:

   There must be a maximum of one OpComm block for each Acquisition Group. The Acquisition Group number is specified in the OpComm mask.

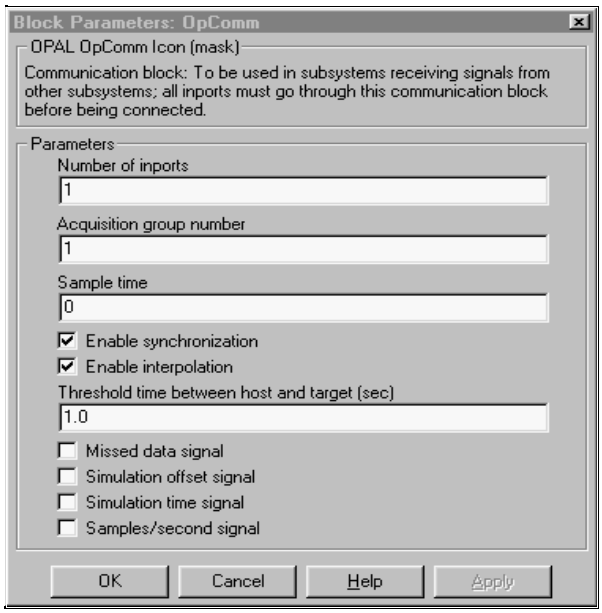
2. When inserting OpComm blocks into a Master (SM_) or Slave (SS_) subsystem:

   There are two kinds of communication associated with target nodes (real-time and non-real-time) but these cannot be sent through the same OpComm block. There must be a maximum of one OpComm block for all real-time communication (communication between target nodes), and a maximum of one OpComm block for all non-real-time communication (communication between the Console and a target node), for a total maximum of two OpComm blocks in any Master (SM_) or Slave (SS_) subsystem.

3.3.3 OpComm parameters for Simulink users

Refer to online help in order to get information about the OpComm parameters.

3.3.4 OpComm parameters for SystemBuild users



In the SystemBuild version of the OpComm, only one parameter is used: the acquisition group number. This is due mainly to the fact that in SystemBuild, the sampling time is specified as a parameter of the superblocks, and similarly that the number of inports (or inputs) is a parameter of the block within SystemBuild.

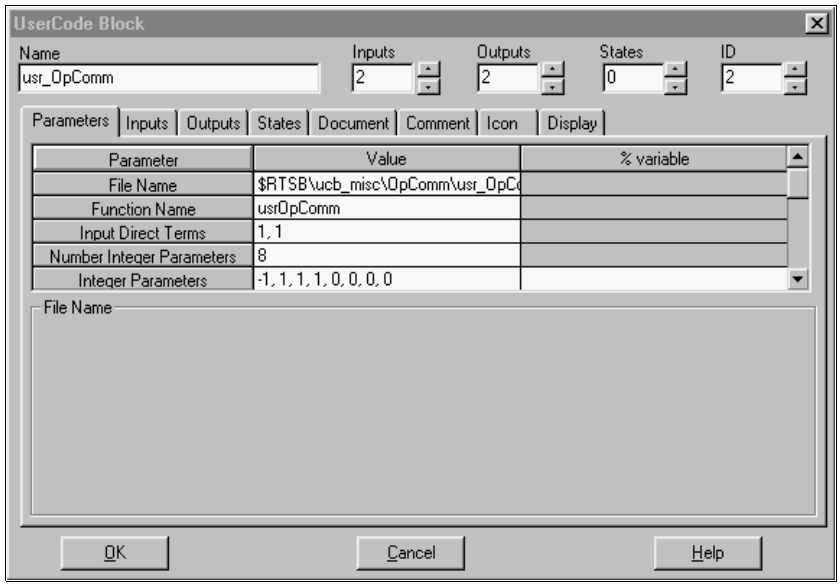


Figure 3-4 SystemBuild OpComm block parameters.

3.3.4.1 Integer parameters for SystemBuild OpComm blocks

There are eight integer parameters, a number of which are illustrated in Figure 3-4. The first is reserved; the others must be specified by the user. These parameters are each set as in the Simulink OpComm block, except *Threshold*, which requires a specific value.

TABLE 3-1 SYSTEMBUILD OPCOMM INTEGER PARAMETERS.

BUTTON / FIELD	FUNCTION
Acquisition Group	This parameter specifies the acquisition group number for all signals that pass through the OpComm block.(Input)
Synchronization	Enables the synchronization algorithm to keep the right signal's shape. (Input)
Interpolation	Enables the linear interpolation between two values during data loss. (Input)
Threshold	(Input)
Missed data	Number of data lost from the simulation platform. Due to network congestion or CPU use, the Console (SC_) is unable to refresh the display with all the data coming from the simulation target platform. In this case, a reception algorithm reacts to the situation to keep the Console synchronized with the simulation target. (Output)
Simulation time	Time elapsed since the beginning of the simulation's execution on the target platform. (Output)
Simulation offset	Time elapsed since the Console started acquiring data from the simulation target platform. (Output)
Samples/second	Number of samples displayed by the Console between two missed data packets. Can also be used to infer average data displayed. (Output)

3.3.4.2 SystemBuild UCBs

When using SystemBuild UCBs in RT-LAB, each UCB must have a minimum of two inputs, and two outputs.

To add output signals:

- Increase the number of outputs on your block by the desired number of signals.

  The number of outputs must be equal to the number of simulation output signals, plus the number of additional output signals.

3.4 Maximizing Parallel Communication

To maximize parallel communication, output data must be exchanged between your network's nodes in the most efficient way possible. The order in which data are sent can be prioritized so that nodes run their main calculations without having to wait from outputs from other nodes in the network. This is accomplished by identifying as state variables those calculations that can be run independent of inputs, and passing input-dependant variables through *Delay* blocks.

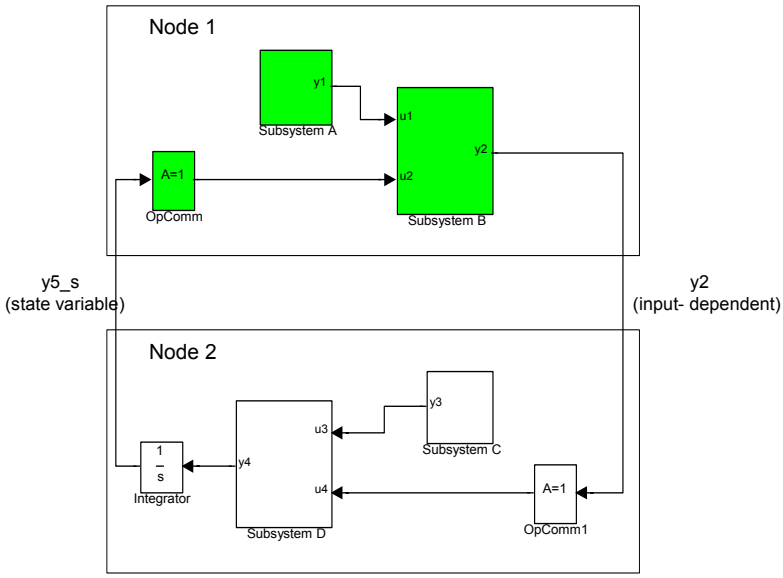


Figure 3-5 This model simulates sequentially since its data are all input-dependant.

3.4.1 Input-dependent data

Figure 3-5 shows the model of a sample simulation on a two-node system. The data exchanged between Node 1 and Node 2 is input-dependent. This means that for any given step, Node 1 must receive its input (*y5_s*) before it can calculate *y2*. Similarly, Node 2 cannot begin calculating *y4* until it has received *y2*.

While Node 1 requires an input before being able to calculate its output, the data exchanged between Node 2 and Node 1 is a state of the system, or a state variable, which means that for any given step, Node 2 does not require an input to determine its output value; only the input from the previous step is necessary. That is to say, it is possible to get the value of *y5_s* at the very beginning of the calculation step, even before *y2* for this step is available, as is illustrated in Figure 3-6.

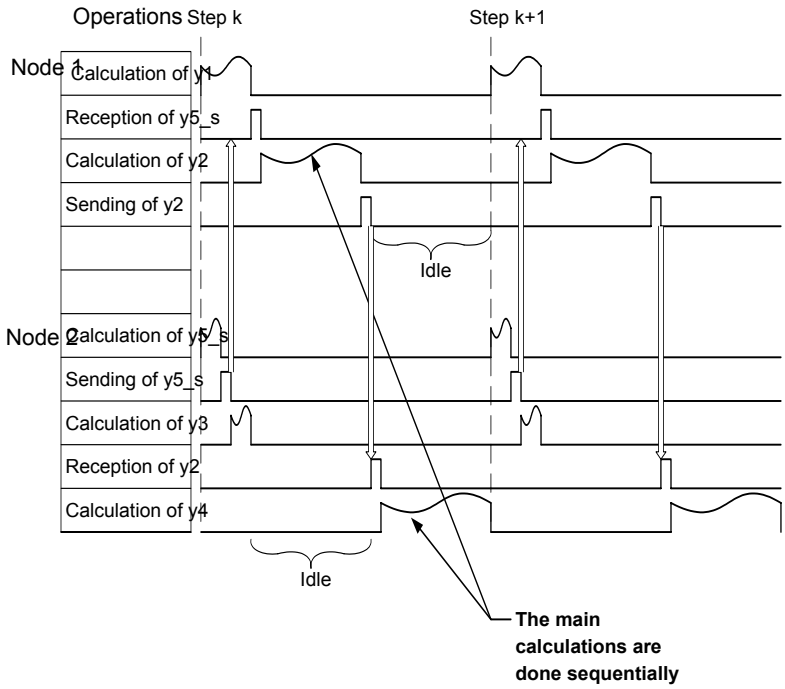


Figure 3-6 Event chart for the sequentially simulated model in Figure 3-5.

Node 2 must remain idle until *y2* becomes available, since Node 2 requires *y2* to make its main calculation. Since *y2* is not a state of the system, it will be sent by Node 1 at the end of its calculation phase. Once *y2* is sent, Node 1 becomes idle, waiting until Node 2 has finished its own calculation phase; once both calculations from both nodes have been completed, a new simulation step may begin.

This example illustrates that when there is an exchange of at least one input-dependent variable between two nodes in a distributed simulation, these nodes will run one after the other (Node 2 must wait until Node 1 has performed its output calculations before it can calculate it own output, and vice-versa). This type of simulation does not take full advantage of RT-LAB's distributed simulation capabilities, or greatly benefit from using a multi-CPU system.

3.4.2 State variables

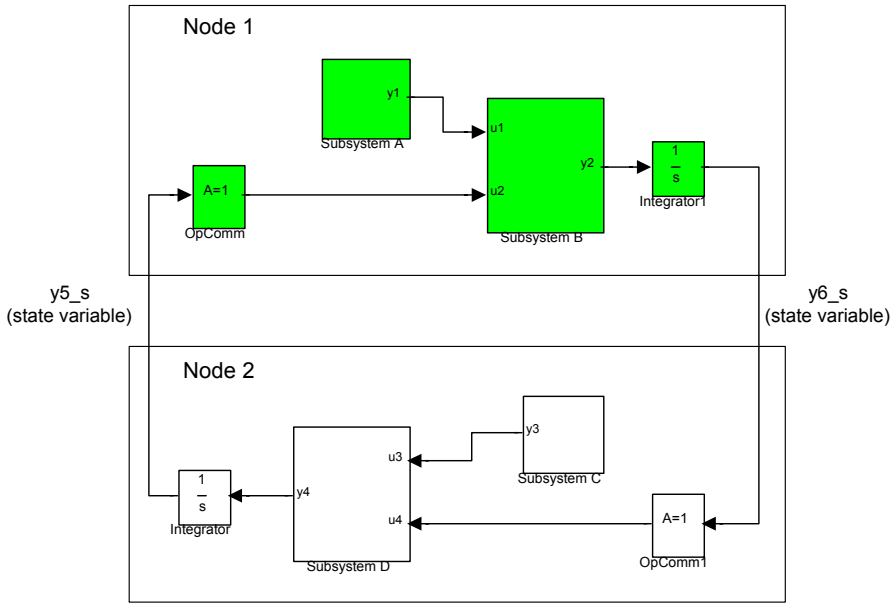


Figure 3-7 Distributed model, including state variables identified to be sent as a priority.

If as in Figure 3-7 Node 1's output is also a state variable (*y6_s*), the data exchanged between Node 1 and Node 2 are states of the system under simulation, which means that it is possible for both subsystems to receive and send this data at the beginning of a given calculation step, as is illustrated in Figure 3-8:

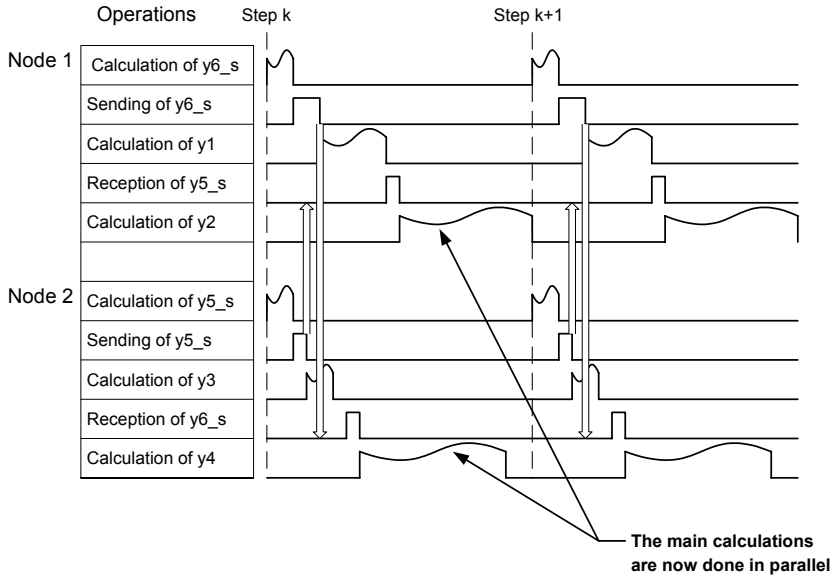


Figure 3-8 Event chart for the distributed model shown in Figure 3-7.

In this example, Node 2 does not have to wait for an input from Node 1 to start its own calculation, since data are now sent at the beginning of the step. The result is that the both nodes' main calculations are done in parallel, and the nodes spend a minimum of time idle. In other words, exchanging state variables between nodes in a distributed simulation will maximize the simulation's parallelism and improve your system's performance.

3.4.3 Delay blocks

While a system being simulated by RT-LAB will have a better performance if all data exchanged between the nodes are in the form of state variables, state variables can also slow your system down. Sending only state variables may, for example, cause a model to perform too many calculations in one node and not enough in another, which would unbalance the distribution and degrade performance. Further, since all data are transmitted simultaneously, if even one input-dependent variable were present in the exchanged data, the model to would run sequentially (see section 3.4.1, Input-dependent data).

The alternative to exchanging state variables is to pass these variables through *Delay* blocks.

If a node sends an input-dependent variable and this variable passes through a *Delay* block, the node will send the value calculated in the previous step rather than waiting for the result of the new calculation. This gives the input-dependent variable the same properties as a state variable: the data can be sent at the beginning of the calculation step.

IMPORTANT! IMPOIRTANT: Using *Delay* blocks will change your system's behaviour.Consider the effects of *Delay* blocks offline, before running your simulation and before operating hardware-in-the-loop.

3.4.4 Key points about network communication

- In a distributed simulation, two nodes will run their calculations sequentially (one after the other) when one or more input-dependant variables are exchanged between them.
- Parallelism is maximized for any two nodes in a distributed simulation when state variables are exchanged between them.
- Parallelism can be maximized for any two nodes in a distributed simulation when input-dependent variables are exchanged between them, by passing each input-dependent variable through a *Delay* block before sending it to another real-time node. (The effect of these *Delay* blocks should first be assessed offline.)

In order to best synchronize communication between the nodes in your system:

- all data sent from Slave subsystems (SS_) to the Master subsystem (SM_) must be sent at the beginning of the step, using either state variables or input-dependent variables passing through a *Delay* block. RT-LAB will generate an error message if a Slave sends an input-dependent variable directly to the Master.

3.4.5 Prioritizing data transmission with Simulink

To exchange all data that are available at the beginning of a simulation step and maximize communication between your network's nodes, the variables' priority must be declared so

that the RT-LAB scheduler will most efficiently send the outputs. In Simulink, this is done by adding an underscore followed by an “s” ("**_s**" as is shown in Figure 3-9) where the “s” stands for *Start by sending*. Adding this notation will instruct RT-LAB to start the calculation step by sending the output data identified by the `_s`.

Figure 3-9 Output Out1 shown on the left is an input-dependent output; the _s added to the Out1 shown on the right declares this output as a priority send.

Note: Adding an “_s” to the end of an output’s label does not mean input-dependent data will automatically become state variables, ready to send at the beginning of the step. Whether an output is input-dependant or a state variable is a property inherent to the nature of that variable in the system, so changing a variable’s name will not affect its behavior. Whenever input-dependant data is to be sent as a priority, a *Delay* block must be used.

3.4.6 Prioritized data transmission in SystemBuild

In SystemBuild, data is automatically sent at the beginning of a calculation step, so there is no need for additional notation to specify a priority send. SystemBuild users must therefore exchange state variables and/or use *Delay* blocks for input-dependent variables.

4 RT-LAB Blocks

RT-LAB comes with a library of blocks that can be added to your model. These allow access to different types of I/O modules and other functions. I/O and synchronization module blocks serve to control external equipment; generic RT-LAB blocks provide other network communication and timing functions. This chapter will discuss the need, use, and parameters of the I/O blocks, synchronization blocks, and other generic blocks that can be used with RT-LAB.

RT-LAB block libraries can be found in the Simulink or SystemBuild library browser.

Note: This chapter is based on the use of IP modules and an IP carrier board, RT-LAB supports other types of boards.For a list of supported I/O boards and for pin-out information for supported I/O boards, go to:

http://www.opal-rt.com/p_rt-lab_devices.html

To add blocks to your model, and set their parameters:

1. Click on the RT-LAB block that corresponds to the desired function of the I/O card you are using.
2. Drag and drop the RT-LAB block into your simulation model.
3. Double-click the RT-LAB block to change its parameters.

   This will open a parameters control box for the block in question.

4.1 I/O and Synchronization Module blocks

I/O modules allow users to control external equipment. These modules may be digital to analog (D/A) converters, analog to digital (A/D) converters, input/output (I/O) binary ports, and quadrature decoders, among others. Synchronization modules synchronize computations and communication between nodes.

Note: I/O blocks have no effect in offline simulation (without RT-LAB).

RT-LAB comes with a library of blocks that allow access to different types of I/O modules. The library modules are listed by their type (ISA, PCI, PC/104, IP-Module) followed by

their vendor name, the product name, and the block. The *RT-LAB_IO* library can be found next to the *RT-LAB* library in the Simulink or SystemBuild library browser.

To find the RT-LAB I/O block library:

- In Simulink, the blocks are located under *RT-LAB I/O* in the Simulink library browser:

  Type **`opal-lib`** in the Matlab command window, then press <Return>.
- In SystemBuild, the blocks are located under *RT-LAB I/O* in the SystemBuild palette:

  Click on the palette, under *RTLab/OpalIO*.

4.1.1 Using I/O blocks in RT-LAB

An I/O module is not independent; it must be inserted onto a supporting board, which usually has two to four module connectors. A supporting board with four ports, for example, could include four A/D converters, or two converter modules, an 8-bit port module, and a D/A converter module. The configuration of each board depends on the application's requirements.

Note: When loading the model, the target subsystem where the I/O block has been inserted (SM_ or SS_) will have to be assigned to the target node with the corresponding I/O card.

4.1.2 I/O block parameters by card type

For each block inserted into a simulation model, different parameters must be set to tell RT-LAB where to locate the card or module that corresponds to the block. The following section describes the different types of cards used with RT-LAB, and the typical parameters associated with each of them.

Parameters specific to the functioning of the board (*Gain*, *Channel*, etc.) are discussed in the section.

4.1.2.1 IP Modules

An Industry Pack or IP module is a small I/O card that connects to an ISA, PCI, or PC/104 IP carrier board. Depending on the model, carrier boards can handle from 2 to 5 IP modules, making IP modules an ideal solution for systems with many I/Os but having a limited amount of ISA or PCI slots on the target node.

IP module block parameters include the carrier board parameters, and the IP module's location. (Carrier board parameters are discussed in the sections specific to ISA, PCI, and PC/104 sections of this chapter.) The IP module's location can be identified by looking at the IP carrier board: the board is labelled with letters from A to E, where *A* corresponds to the first IP slot, *B* corresponds to the second, and so on.

4.1.2.2 ISA card

ISA cards are addressed by specifying the base address of the board, which is usually configured by jumpers or a DIP switch. Refer to the I/O card's user manual to set or get the value of the base address, or for a description how to set the value for the *interrupt line number*.

Note: ISA cards are usually manually configured by the user; ensure there is no address conflict with other cards in the computer.

4.1.2.3 PCI card

PCI cards already posses unique Vendor and Product ID numbers, which allows them to be easily found in the target computer. The PCI cards' card address and Interrupt line numbers are automatically assigned by the PC, which ensures there is no conflict with other cards.

In cases where multiple cards of the same model (having the same Vendor and Product ID) are found in one computer, the PC will assign each card a different index, often called the PCI index. When only one card is inserted in the PC, it will always have a PCI index of zero (0); when two cards are inserted, one card will be assigned a PCI index of zero (0) while the second card will have a PCI index of one (1). The PCI index ordering usually starts from the card closest to the processor, but since some motherboard manufacturers do not follow this rule, the best way to know which card has index-zero and which card has index-one is to test the cards. To do so, make an RT-LAB simulation model with blocks for both index-zero (0) and index-one (1) then test which one is working when connecting only one of the two cards external connector. The same technique can be used when more than two cards are used.

4.1.2.4 PC/104/104 card

Since the PC/104 bus is actually an ISA bus with a different connector type, the same ISA bus addressing scheme is used. Refer to the ISA section for more details on PC/104 addressing.

4.1.2.5 I/O Synchronize option

By default, the RT-LAB I/O blocks are executed as they would be in Simulink offline mode: the blocks are executed one after the other. Adding flowcharts, enabled subsystems, or other blocks causes computation times to vary. The time interval between two calls to an I/O block will vary as well, depending on the computation time of the blocks that were called first. This computation variation will result in an error since the time between two I/O card accesses will not be constant.

By using the RT-LAB I/O block's Synchronize option, I/O hardware will be called before any other blocks are called, to ensure constant I/O access intervals at each step, and eliminating the error.

4.2 RT-LAB I/O and Synchronization Module blocks

The block library includes blocks for every available type of I/O module. Each block described in this section is available for each card type, and the card type will be identified in the block's name: while for the purposes of this manual an block will be referred to as *OpAnalogIn*, for example, the block library will list the block as *Op*XYZ*AnalogIn*, where XYZ is the card name applicable to your system.

The I/O and synchronization module blocks discussed are:

- *OpAnalogIn*
- *OpAnalogOut*
- *OpDigitalIn*
- *OpDigitalOut*
- *OpQuadDecoder*
- *OpSync_VME*

4.2.1 OpAnalogIn

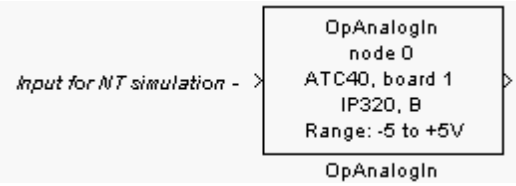


Figure 4-1 OpAnalogIn block.

This block is used to receive the value of an analog input. The signal received by this input is displayed at the block's output. The information is quantified through an A/D converter. The basic channel is always channel 0. To capture the signals on numerous channels, a de-multiplexing block must be inserted at the block output. The signals will then be extracted at the output in the same order as the connections for the de-multiplexing block, with the signal at the top of the de-multiplexing block being the first one (Channel 0).

TABLE 4-1 OPANALOGIN I/OS

Input / Output	Function
Input	Input is used only for the simulation. It requires the value of the input signal in volts. This input is directly connected to the output, without the quantification error that can be generated in an analog-to-digital conversion.

TABLE 4-1 OPANALOGIN I/OS

Input / Output	Function
Output	The output block corresponds to the voltage value of the signal present at the input of the physical module. In simulation mode, this output is directly connected to the input.

4.2.2 OpAnalogOut

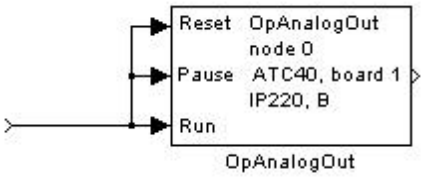


Figure 4-2 OpAnalogOut block. The module in this figure would maintain its last output value when paused and reset, since their inputs have the same value as the *Run* input.

This block corresponds to a physical module with an analog output. When simulating, signals sent to this block will be routed to the analog output module located at the specified address. The signal applied is also sent to the block's output for use within the model. Depending on which module is used, the information is quantified and sent to a D/A converter. The base channel is channel 0. To use multiple channels, insert a multiplexing block before the inputs: all signals sent to the multiplexing block will be sent to the module. At the output, these signals will be in the same order as they are in the multiplexing block, with the signal at the top of the block being the first channel (channel 0).

TABLE 4-2 OPANALOGOUT I/OS

Input / Output	Function
Reset	(Input) This input should be connected. It is the value to be transmitted at the output when the system is reset (brought back to zero) whether internally, or using the Probe Control Panel. If numerous signals are sent, a multiplexing block must be inserted, in order to enter values for each channel. Any channel without a given value will be initialized with a value of zero.
Pause	(Input) This input should be connected. It is the value to be transmitted at the output when the system is paused. If numerous signals are sent, a multiplexing block must be inserted, in order to enter values for each channel. Any channel without a given value will be initialized with a value of zero.

TABLE 4-2 OPANALOGOUT I/OS

Input / Output	Function
Run	(Input) This value corresponds to the desired voltage at the converter's output. Keep in mind that the physical output will contain a quantification error due to the digital to analog conversion. This input can be connected to the *Reset* and/or *Pause* inputs to maintain the last output values when the system is reset and/or paused.
Output	None.

Note: Connecting a simple constant to the *Pause* and *Reset* inputs will not make this value apply to all channels. This value will apply only to the first channel, and all other channels will take the default value (0). Use a multiplexing block to define *Reset* and *Pause* values for all channels.

4.2.3 OpDigitalIn

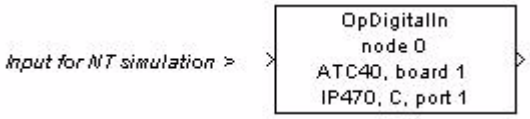


Figure 4-3 OpDigitalIn block.

This block is used to read the bit values of a physical module port. The output value corresponds to one bit of the port. To read all 8 bits of the port, a de-multiplexing block must be inserted at the output.

TABLE 4-3 OPDIGITALIN I/OS

Input / Output	Function
Input	Since it is directly connected to the module output, the input is used only during simulation. It corresponds to the value of one bit of the input port. A multiplexing block can be inserted before the block to simulate the 8 bits of a port.
Output	The output corresponds to the value of one bit of the physical port chosen.

4.2.4 OpDigitalOut

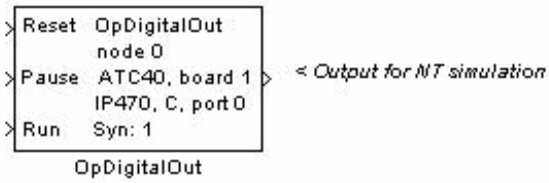


Figure 4-4 OpDigitalOut block. OpDigitalOut maintains its last output value with *Pause* and *Reset*.

This block allows data transmission through a physical module's output port. Each block allows the transmission of bits 0 to 7 of a port.

TABLE 4-4 OPDIGITALOUT I/OS

Input / Output	Function
Reset	(Input) Value to be transmitted at the output when the system is reset (brought back to zero). If more than one bit is transmitted, a multiplexing block must be used to enter the value for each bit. Any bits without a given value will be initialized at a value of zero (0).
Pause	(Input) Value to be transmitted to the output when the system is paused. If more than one bit is transmitted, a multiplexing block must be used to enter the value for each bit. Any bits without a given value will be initialized at a value of zero (0).
Run	(Input) Corresponds to the value of one port bit, and must be either zero (0) or one (1). This input may be connected to the *Reset* and *Pause* inputs to keep the old output values when the system is reset or paused. Inserting a multiplexing block enables the writing of all bits to the same port.
Output	The NT simulation output is the result of the quantification of one bit. An input value greater than 0.5 will yield 1 as its output value; a value less than or equal to 0.5 yields an output equal to zero (0).

4.2.5 OpQuad-Decoder

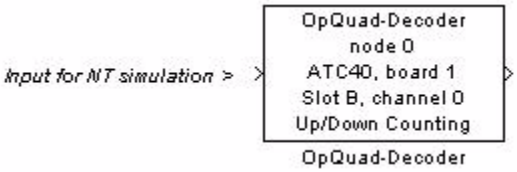


Figure 4-5 OpQuad-Decoder block.

This block enables reception of the value from the quadrature meter. This value is transposed into a signed number, and generally corresponds to a position.

TABLE 4-5 OPQUAD-DECODER I/OS

Input / Output	Function
Input	Used for the simulation, this input corresponds to the value from the quadrature meter. Since the meter has 24 bits, the values must therefore vary between + 8 388 607 and – 8 388 608.
Output	The value from the quadrature meter. This is a signed value. In the simulation, it contains the same value as its input.

4.2.6 OpSync_VME

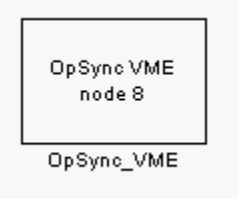


Figure 4-6 OpSync_VME block.

This block is inserted by the user into a target (computation) node subsystem, to synchronize the acquisitions of the ATC-40 boards, with the help of the SEDERTA SD-VME-200 data acquisition board located in the VME casing.

TABLE 4-6 OPSYNC_VME I/OS

Input / Output	Function
Input	None.
Output	None.

5 Acquiring and Viewing Data

This chapter discusses aspects related to the acquisition and viewing of your simulation data, including acquisition groups and their parameters, and how to determine the speed of your simulation and optimize its synchronization. Inter-node communication is also discussed, both between target nodes and the Console, and between target nodes across FireWire or cLAN.

5.1 Acquisition Groups

RT-LAB allows signal acquisition from the real-time target nodes in the cluster. Because TCP/IP is relatively slow compared with most real-time systems, RT-LAB allows users to set different groups of acquisition. Each group can include many signals, including the number of signals per frame to receive, decimation factor, etc. These parameters can be defined separately for each signal. For example, in a system with a 1 millisecond step, a user might want to acquire the data values of a slow changing signal once every 100 steps or so. Signals can also be acquired only once specific conditions have been met, by defining the conditions' parameters in a Trigger block, as described below.

Signals received on the Console (SC_) subsystem are grouped into twenty-five groups. The acquisition group for a signal is specified in the OpComm communication block through which the signal passes.

5.1.1 Acquisition parameters

Acquisition parameters are shared by all signals included in a given acquisition group. These parameters are specified by the user in the Probe Control Panel, on the tabs numbered with the appropriate acquisition group number.

For more details on how to control data acquisition parameters, see online help for Probe Control Panel.

5.1.2 Triggering Acquisition

By default, each acquisition group is always triggered, meaning that a new data packet is filled as soon as the previous packet is sent. Sometimes, however, you may want to receive data only when a certain condition occurs.

For this reason, the *OpTrigger* block can be inserted into the subsystem containing the acquisition group signals. The *OpTrigger* block is used to acquire data triggered by a specific signal, allowing users to acquire information from a specified parts of the simulation only. For complete details about *OpTrigger* and its parameters, see the online help.

5.1.3 Recording data with OpWriteFile

To obtain all data for any particular signal, use the *OpWriteFile* block. Data recorded with the help of this block will be stored on the QNX real-time node's hard drive. The user will have to transfer the data onto the NT station after the simulation stops: the data transfer is not automatic.

Sometimes, in simulations with fast sampling times, the acquisition data loss on the NT is too significant to clearly understand what is happening in the real-time model. The Simulink library *ToFile* block cannot be used in real-time simulations since it does not have a buffering system. Using this block in a real-time context without a buffer would significantly increase the real-time simulation's effective step size.

A real-time write-to-file block (*OpWriteFile*) can be inserted into the computation subsystem. This block allows data in MATFILE format to be saved to the real-time node's hard disk. The generated MATFILE (.mat file) can be transferred to the NT workstation, and analyzed using MATLAB or Xmath with a mathscript file (<RT-LAB Installation dir> / SystemBuild/lib/readOpFile.msc). The write-to file does not affect the simulation step size in synchronized mode, since all writing is done during the simulation's spare time. The *OpWriteFile* blockblock uses a circular buffer architecture that allows continuous data acquisition. If the circular buffer is full, it will be emptied completely before starting again. For simulations with small step size, a large buffer size is recommended to avoid discontinuous data in the write-to file.

The Simulink block has one input: the signals to be saved. You can save multiple signals by using a vector as the block input. In SystemBuild, the *OpWriteFile* block should have as many inputs as there are signals to record. Each *OpWriteFile* block must be associated with an acquisition group number. Also, since the write-to-file block is considered an extra acquisition group, an *OpTrigger* block can be used to write data only when certain conditions occurs.

For complete details about *OpWriteFile* and its parameters, see online help

5.2 Understanding Data Reception

5.2.1 Communication between the Console and the Target nodes

The Console (running on Windows NT) is connected to each computation node (in the QNX environment) by an Ethernet network. Since this type of network is much slower than a FireWire-type network, and since an NT workstation is very resource-demanding, the NT workstation will probably not be able to receive and display all of the data, so there is a need to be selective with respect to incoming data. Often, data is read only every 2, 3, or even 5 calculation steps: this lightens the load imposed on the Console. In order to accelerate the transmission process, target nodes send batches of data to the network, rather than sending one batch per data item. The target node contains a buffer, where it gathers data from many signals for transmission to the Console. The target node will dispatch its data only when this buffer reaches the level specified in the Probe Control Panel.

A frame is composed of all of the data that represents a specific signal inside the buffer. For example, if you have three signals and want the data transmitted at every 5000th value, the buffer will contain 15,000 samples, or three frames of 5000 values each. Because the NT workstation takes a certain amount of time to receive and display the acquired values, and because the real-time model is calculated in continuous mode, it is possible that the next data frame received may not immediately follow the currently displayed frame; in other words, there may be "holes" in the data display. The following figures illustrate this point:

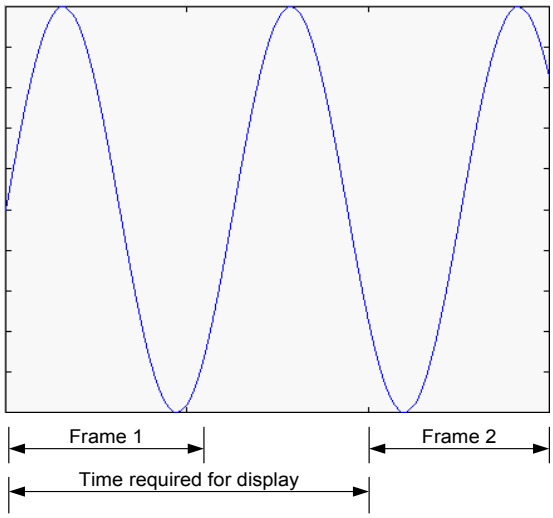


Figure 5-1 Expected signal in the data display.

As the time required to display Frame 1 is longer than the frame itself, the portion of the calculation that is performed between dispatching the two frames will be lost, creating a discontinuity in the display, as shown below:

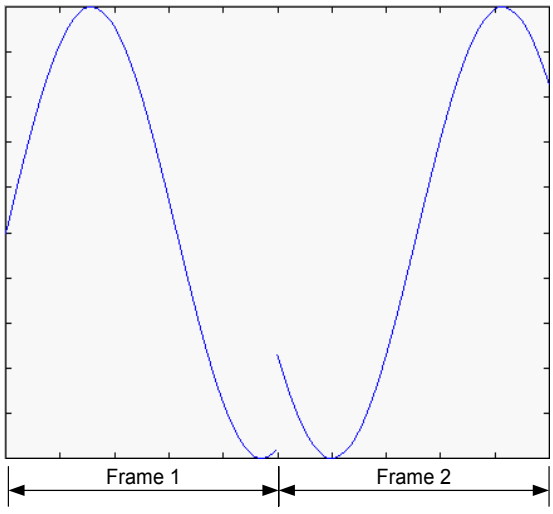


Figure 5-2 A "hole" is generated in the data display.

It is possible to minimize this effect. For a number of values per given signal, an increase in the decimation factor (from one acquisition per step to one acquisition for every two steps) also increases the effective length of the frame, diminishing the gap in values between the two data transmissions:

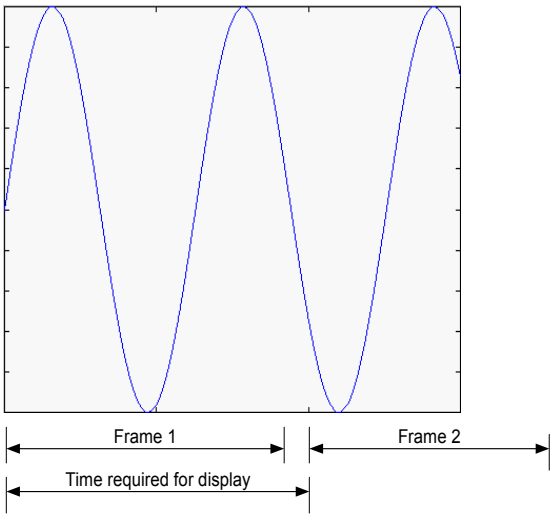


Figure 5-3 Increasing the effective length of a frame by increasing the decimation factor.

Although the display will be more representative of the real-time calculation performed, an increase in the decimation factor can cause problems with display resolution:

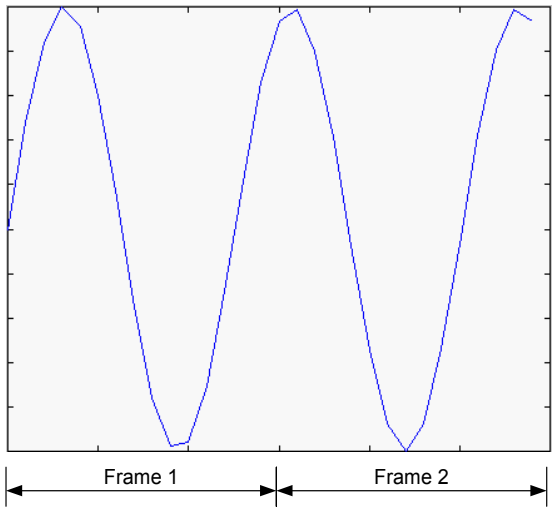


Figure 5-4 Display resolution error that can result from increase in decimation factor.

To synchronize all tracings with one another, be sure that all signals you want to view pass through the same target node.

In RT-LAB, even with an NT console crash, a real-time model running on the QNX nodes will continue running. After rebooting NT, reconnect to the real-time model and continue data acquisition. See online help for Main Control Panel, for more information.

5.2.2 Synchronous Acquisition

Remember that acquisition frames are sent when the system has time. Frames sent as TCP/IP packets will be sent while the model is waiting for a FireWire packet or for the synchronization pulse. This results in your real-time system not being disturbed by a TCP/IP transmission. As a result, the NT display could miss some data if the Console station is too slow, or the QNX node does not have enough time left. This technique helps ensure hard-real-time performance.

Aliasing is another acquisition issue, which arises with the use of small frames. As was explained above, it is possible that the next data frame received may not immediately follow the frame currently displayed. Consequently, in case of small frames (like one value per frame) the possibility of losing some values results in incorrect data display. To illustrate this concept, let's assume that we have a sine signal as in Figure 5-5.

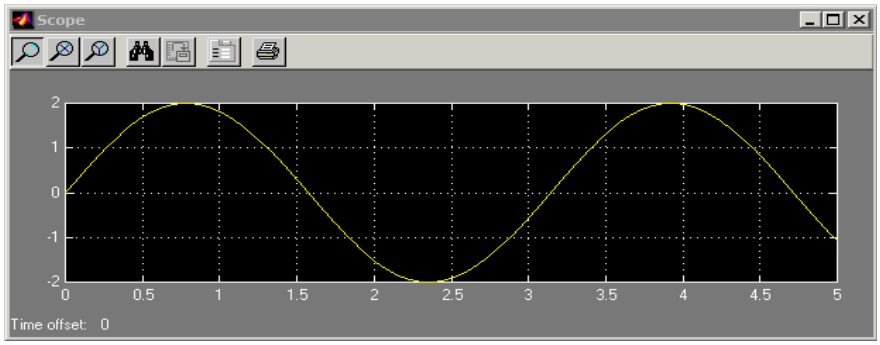


Figure 5-5 Signal with no aliasing.

If, for example, data is acquired once every *n* steps because acquisition cannot follow the real-time simulation, the display will show values without accordingly updating the time axis, which results in aliasing: the frequency of the resulting signal will be increased by a factor of *n*.

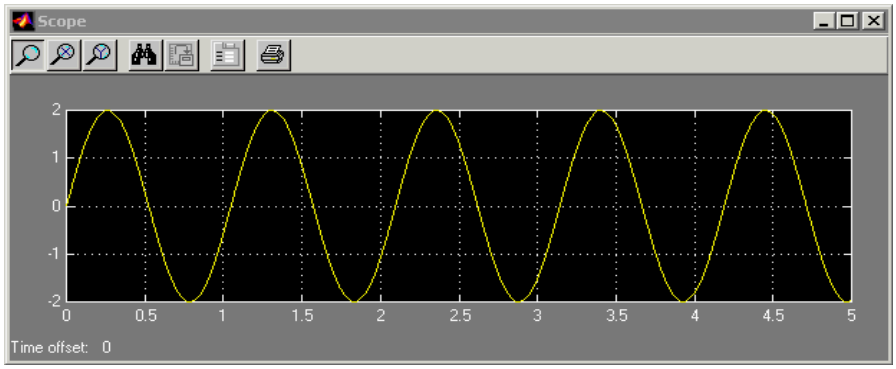


Figure 5-6 Sine waves showing aliasing due to a non-synchronized time axis.

The data reception problem occurs because the Console isn't aware that data from the target node is being missed. To circumvent this problem, it is possible to synchronize the Console time with the simulation time to avoid data loss and data reception errors.

During acquisition, the console subsystem acquires signals from the simulation platform, allowing the user to view data (*Scope*, *To Workspace*-type blocks, etc.). As was mentioned in *Chapter 3, Building a Distributed Model for RT-LAB*, users can choose three options from the OpComm block mask, as outlined in Table 5-1.

Note: In the following options, interpolation suggests estimating the value between two known values.

TABLE 5-1 OPCOMM ICON MASK OPTIONS

BUTTON / FIELD	FUNCTION
Enable synchronization	Enables the synchronization algorithm for a given acquisition group. This algorithm should be used if synchronization is necessary between acquisition Console time and simulation time. In case of a synchronization problem, the Console will detect missing data from the target node, and the algorithm will either replace the lost data with the last signal value received, or will interpolate the data.
Enable interpolation	In case of missed data from a computation subsystem, the synchronization algorithm can interpolate data during a range of data loss. The two images in Figure 5-7 illustrate this point: The display at left shows no interpolation during data acquisition; the display at shows interpolation between two frames during data loss.
Threshold	Difference between the simulation Console time and the simulation target time. Whenever this difference is exceeded, the synchronization algorithm stops interpolating and re-synchronizes to the new value.

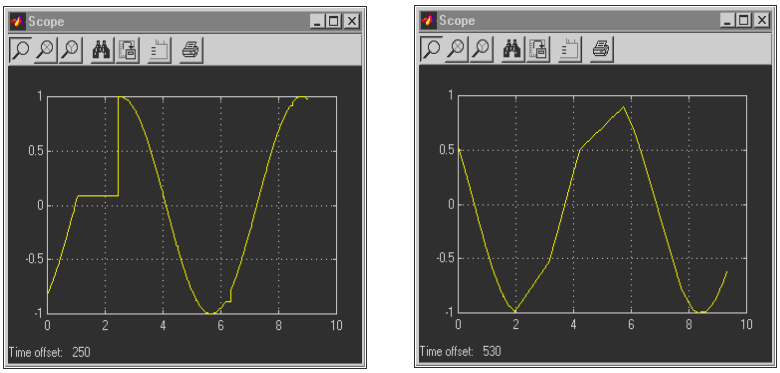


Figure 5-7 Interpolation in data acquisition.

6 Models with Multiple Sampling Rates

RT-LAB supports multirate models—that is, models that have multiple step sizes—by using the fundamentals described in the multirate RTW and Simulink documentation. Before continuing with this chapter, it is strongly recommended that you read *Chapter 7* of the *Real-Time Workshop User's Guide*, where the basic concepts of multirate models are explained. The *Real-Time Workshop User's Guide* is located in the following file:

C:\MATLABR11\help\pdf_doc\rtw\rtw_ug.pdf

Note to Simulink Users: Simulink documentation uses the terms *sample rate*, *sample time*, and *step size* interchangeably. To avoid confusion, we use sample time for the period of analog-to-digital conversions during data acquisition, and step size for the period of model computation, and sample rate and step rate, which are their inverses.

RT-LAB supports three types of models with multiple sampling rates:

1. Multithread, Single-CPU;
2. Single-Thread, Multi-CPU;
3. Multithread, Multi-CPU.

Each of these options is discussed in this chapter:

6.1 Requirements for running models in Multithread, Single CPU mode

To run your model in Multithread mode, insert an *OpSync* icon in the Master subsystem (SM_) and run your simulation in *Synchronized (Hardware/Software)* mode, as specified from the Main Control Panel. Unless these two conditions are met, the model will run in Single-Thread mode, meaning that only one process will handle all of the model's sampling rates.

To select Multithread mode:

1. Open Simulink's Simulation Parameters window, and click on the *Solver* tab.
2. Click *Fixed step*.
3. Specify a *Fixed step size*, and set the Mode to *Auto* or *Multitasking*.
4. Click *Ok.*

.

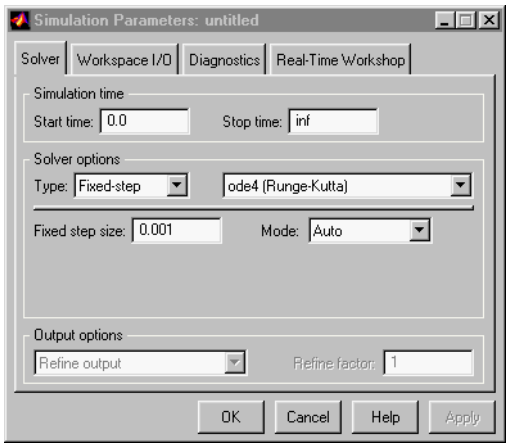


Figure 6-1 Specifying a Fixed step of 0.001 with Multitasking.

6.1.1 Determining base sampling rate

The base step size of the Master subsystem (SM_) is determined according to the value in the *Fixed step size* field (0.001 seconds in the above example). All other step sizes are determined according to the step size parameters specified in blocks that use this information, such as integrators, delays, etc.

6.1.2 Making proper rate transitions

In order to account for transitions between fast and slow rates and vice-versa, ensure that *Zero-Order Hold* and *Delay* blocks are inserted according to the concepts illustrated in pages 7-11 to 7-15 of the *Real-Time User's Guide*.

6.1.3 Compiling and Loading the model

Once the *Zero Order Hold* and *Unit Delay* blocks are added, the model must be compiled and loaded from the Main Control Panel. The following message should be displayed in the Phindows or RT-LAB display window:

```
Real-Time Multitasking mode (Multi-Thread)
```

If this message is not displayed, make sure the *Hardware Synchronized* or *Software Synchronized* option is selected in the Main Control Panel. Messages about processes associated with the different rates are also displayed. For example:

```
Executing baseRate(0.001), pid=160. (prio=27)
Executing subRate(0.002), pid=154. (prio=26)
Executing subRate(0.004), pid=156. (prio=25)
```

For the purposes of this discussion, we will refer to the above three messages as A, B, and C. Message A has the highest priority, and cannot be interrupted by any other process. Messages B can be interrupted by Message A, since it has a lower priority level than A, but can not be interrupted by Message C. Message C, in turn, can be interrupted by both Messages A and B since it has the lowest priority level.

6.2 Requirements for running models in Single-Thread, Multi-CPU mode

In Single-Thread multirate simulation, the Master subsystem (SM_) has one fixed sampling rate, which is specified in the simulation's Parameters window. The Slave subsystems (SS_) are all at the same rate (consequently, all *OpComm* blocks contained in the Slave subsystems must have the same sampling time) but this rate is different from the Master subsystem's rate.

6.2.1 Inserting blocks for proper rate transitions

Single-Tasking mode (specified in the *Solver* page of the Simulink Simulation Parameters window) does not require users to insert *Zero Order Hold* and *Unit Delay* blocks. However, it is recommended to insert these blocks even if they are not required, to allow for the possibility of running the model in Multitasking mode (see below).

The *OpComm* in the Master subsystem must have the same sampling time as the Slave *OpComms*. Even though the Master subsystem runs at a different rate, this rate is determined by the fixed step size in the Simulink model's simulation parameters. The value in the *OpComms* defines the communication rate, which is the rate of the slowest node in the system, that of the Slave subsystems.

6.2.2 Communication rate

In the example shown in Figure 11.2, the Master runs at a step size of 1ms, and communicates with the Slave subsystems once every 10 steps, while the Slave runs at a step size of 10ms, and communicates with the Master subsystem at this rate. The Master holds and uses the same input values for its intermediate steps, so the communication rate is 10ms.

The following message should appear in each of the Phindows or RT-LAB display window when you load the model:

```
Common sample time = 0.010000

Non real-time Single-Tasking mode.
```

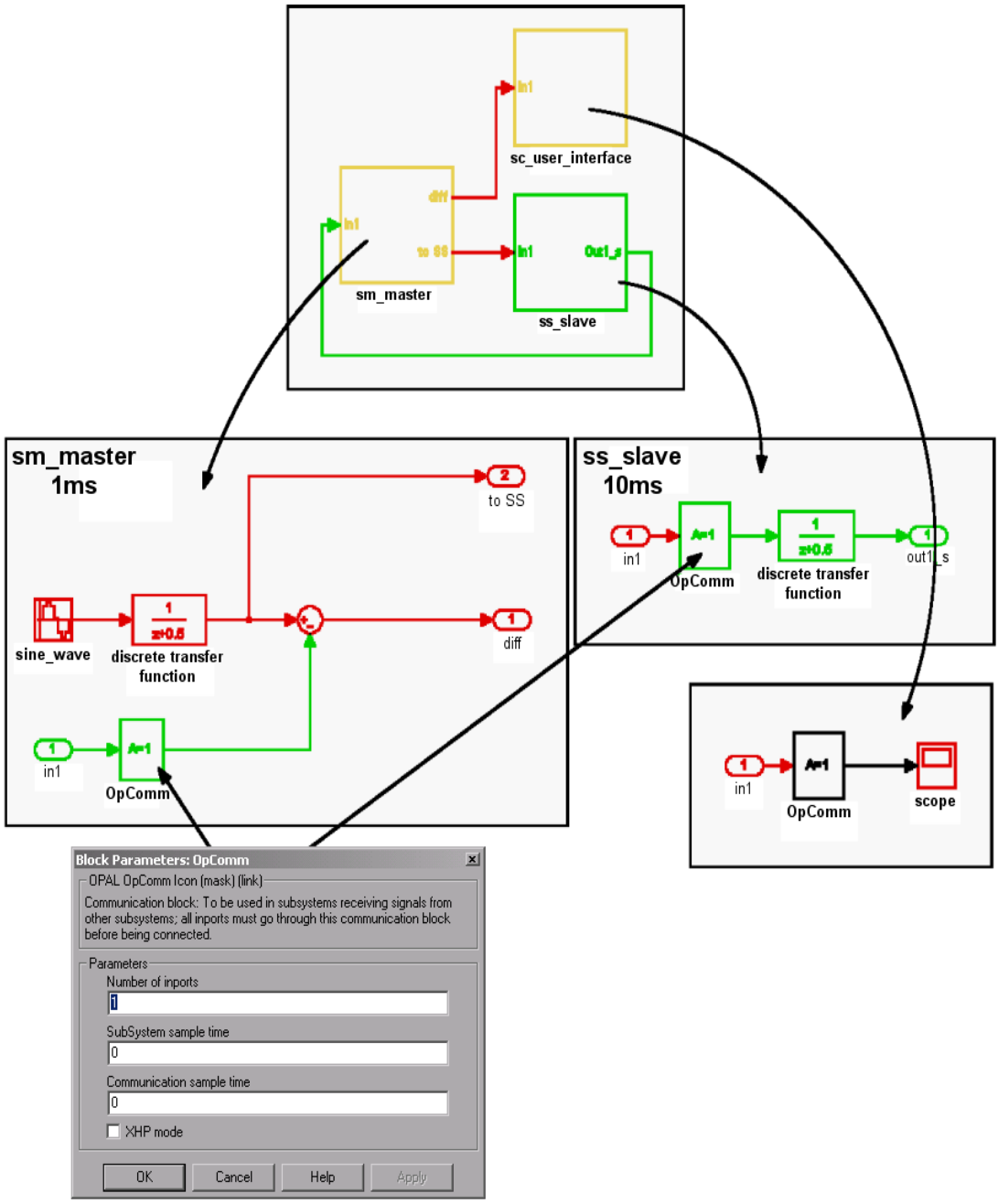


Figure 6-2 The Master runs at 1ms and the Slaves run at 10ms in this model.

6.3 Requirements for running models in Multithread, Multi-CPU mode

Multithread, Multi-CPU blends the previous two cases. Both the Master and Slave subsystems can each run at multiple step sizes. All of the model's OpComms must have the same rate, equal to the communication rate. This rate is defined as the fastest rate which the Master subsystem can use to communicate with all subsystems to which it is connected.

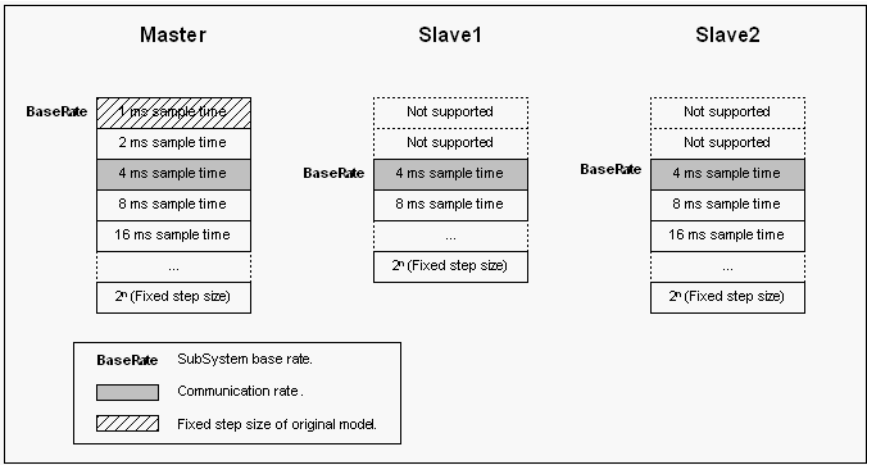


Figure 6-3 Multithread, Multi-CPU: possible sampling times in a model that would run on 3 CPUs.

In the above example, the *Fixed Step Size* in the Simulink Simulation Parameters must be 0.001sec, and the mode must be set to *Auto*. Multitasking mode is not recommended for this type of simulation. Even if the original model is Multitasking, it is possible that the subsystems will require execution in Single-Tasking mode if any of them only holds one rate. This situation would generate the following errors at the model separation stage.

```
??? Error using ==> make_rtw

Error using ==> rtwgen

Model 'Name' does not contain multiple sample times. You
must specify solver mode as Auto or Single-Tasking.
```

In this example, the OpComm blocks connecting the Slaves to the Master must have their sampling times set to 0.004. As in the first case, *Zero Order Hold* and *Unit Delay* blocks should be inserted to properly accommodate all rate transitions inside a given subsystem.

Note: Slave subsystems cannot have rates faster than the communication rate. This is due to the fact that the Master subsystem is synchronized using a timer card, while the Slaves are synchronized using communication with the Master subsystem.

6.4 Multirate mode options

6.4.1 Using an SC_ Subsystem in Multirate Mode

Since the Console is executed in Simulink mode on the NT station, the concept of Multithread is irrelevant to the SC_ subsystem. The user must be careful to properly insert the *Zero Order Hold* and *Unit Delay* blocks in order to achieve communication between the Console and the real-time subsystems.

For any given subsystem receiving data from the Console, the OpComm connecting the two must have the *Fixed step size* rate, the rate at which the Console operates. This rate has to be the same as the communication rate.The *Fixed step size* can be imposed by setting the OpComm block's *Sample Time* parameter to 0. Depending on the computing subsystem's rates, transition from a faster rate (Console's *Fixed step size*) to a slower rate (the computing subsystems' rates) must be taken into consideration.

Similarly, transition from slow to fast rates must be considered for data sent from a Master or a Slave subsystem to the NT workstation for display via the Console.

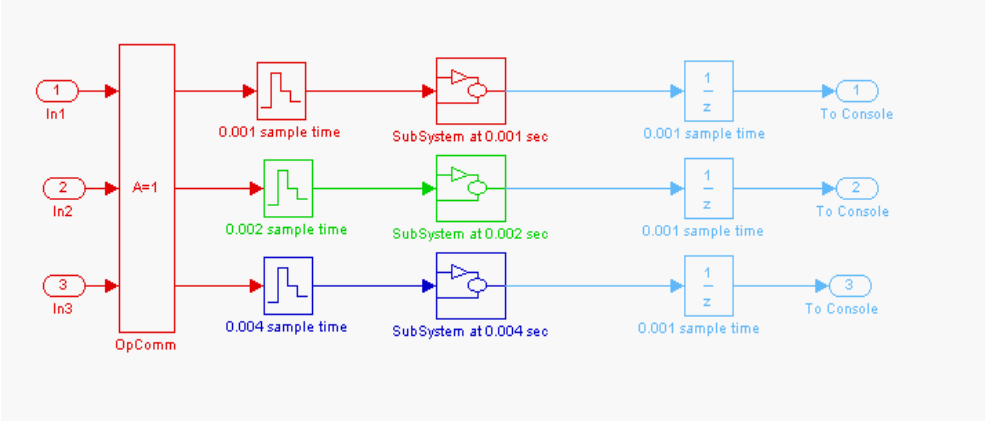


Figure 6-4 Rate transition for sending data to the Console subsystem.

6.4.2 Using I/O blocks in Multirate Mode

When using I/O blocks in a multirate model, it is strongly recommended to place them in the Master subsystem (SM_). Since it is synchronized using a hardware timer card, the Master subsystem offers the most reliable environment for I/O operation.

Figure 6-5 shows that even if the sine wave is generated at a rate of 0.005sec, data transmitted to the hardware I/O through the *OpIP220AnalogOut1* are triggered at the *Fixed step size* rate of 0.001ms. This has the effect of a zero-order hold: data sent to the I/O will have the same value for 5 steps until the sine wave's value changes.

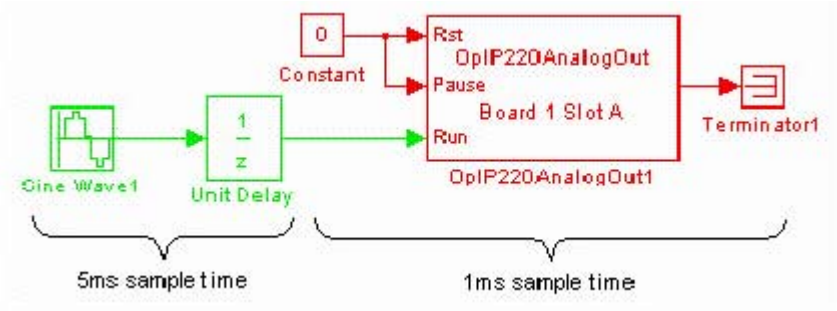


Figure 6-5 I/O in Multitasking mode.

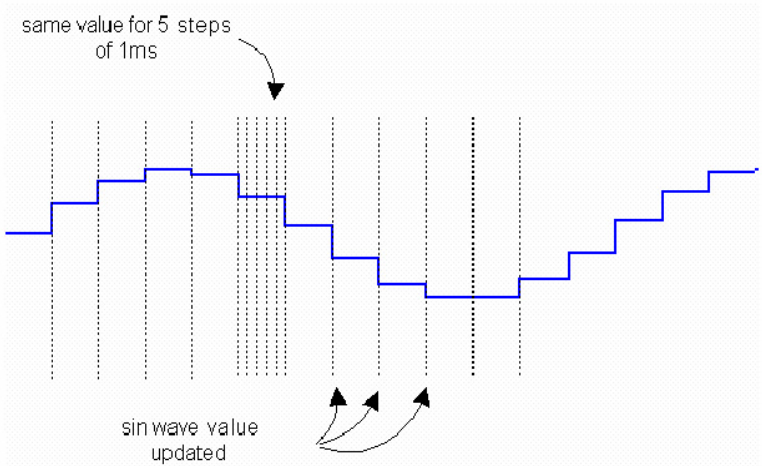


Figure 6-6 I/O executing at the base rate of the Master.

7 Running Your Simulation

After distributing the model and generating its associated C code as described in the previous chapters, next in the model implementation process is compilation of the model, and finally its simulation. This chapter explains the procedures involved in compiling, simulating offline, loading your model, executing its simulation, and interacting with the model during simulation.

7.1 Compiling

Compiling your distributed model is a fully automated process, initiated by clicking on the *Compile* button on the Main Control Panel. A dialog box will present four check boxes to be completed before you proceed. Right-click *Compile* to reset the parameters.

When you click *Compile*, RT-LAB will:

- Separate the Model into the SC_, SM_, and SS_ subsystems you defined.

  The now-grouped block diagram will be split into smaller diagrams, each associated with a different processor. There will be one diagram generated for each SC_, SM_, or SS_ subsystem.

- Automatically generate code on the Master and Slave computers.

  The Simulink sub-models are coded into C language by the MATLAB Real-Time Workshop (RTW) module. RTW also creates a makefile according to a specified template. Since real-time models are executed under the QNX environment, the template used is one designed for compiling under the QNX operating system.

  The ISI C code generator, AutoCode, is used to translate SystemBuild models into C code. RT-LAB provides template files for AutoCode when generating C code for SystemBuild models. It also modifies a template makefile according to the C program generated, to be used when compiling under the QNX operating system.

  During execution of the model's real-time simulation, the SC_ model can interact with the simulation, since C code is not generated for the Console.

- Automatically compile the C code

  The sub-model files, now coded in C, are then compiled within the QNX environment. Files required for compiling are transferred by Ethernet link from your NT workstation to a workstation operating under QNX. This QNX station compiles all C files for the sub-models, to generate files ready for execution on the target nodes.

- Automatically apply panel settings

  After the code is generated and the model compiled, the user must set execution options before running the simulation. These options, discussed in Chapters 6 through 10, deal with assigning the model to physical CPUs in the network, and setting the configuration parameters through the Configuration panel.

7.2 Simulating Offline

Models can be simulated offline, that is within their Simulink or SystemBuild environments, even after subsystems have been defined and communication blocks inserted.

This option is particularly useful for isolating various problems that may exist within the model. During real-time execution, problems can arise at many levels in the system, making it hard to determine the exact source of the problem. By running the grouped model's simulation off-line, you can investigate any results associated with delays caused by parallel communication, or by the "sample and hold" of I/O communications, without having to worry about problems related to the peripheral hardware (communication boards, memory allocation, bad connections, etc.). Additionally, those using RT-LAB in a HIL context can modify and check the command system offline, which reduces the risk of damage to the hardware.

7.2.1 Loading your model and executing its simulation

To start simulation execution:

1. Start the Main Control Panel by double-clicking on the *MainControl.exe* icon on your desktop.
2. Select the Simulink or SystemBuild file for the grouped model (the .mdl or .sbd file) by clicking on the *Open Model* button.
3. Select the target platform, and then click on the *Compile* button in the Main Control Panel. This automatically starts the following processes:

   - separation of the grouped model into subsystems;
   - C code generation for each subsystem;
   - code compilation for the various subsystems for real-time execution.

   At the end of this step, an executable file is generated for each subsystem of the model. Each file is executed by a target node as previously assigned.
4. Assign the generated executables to the real-time cluster's physical nodes. Click on the *Assign nodes* button *to* bring up the list of executables. These constitute the distributed model, and the list of available computing nodes. Each executable can then be assigned to a node in the system.
5. Load the executable files by clicking on the *Load* button; if the *Load/Reset Console Also* option is checked, the MATLAB/MATRIXx program starts automatically, and the Console subsystem file is displayed on screen. The RT-LAB display window will appear for the various system nodes, displaying simulation diagnostics between the nodes.
6. Click on *Execute* to start the system.

7.2.2 Interacting with the model during execution

You can interact with the model and change its parameters during real-time execution via the Console subsystem, located on the Command Station.This feature is useful for adjusting parameters that control model performance: gains, time constants, various limits, etc.

Parameters modified through the Console will affect the simulation only after a certain amount of time has elapsed. The length of this delay depends on the importance of the load handled by the TCP/IP connection linking the Console to the target nodes. This delay is typically of the order of milliseconds, which is negligible compared with delays associated with human-in-the loop interaction in such systems. Nevertheless, when modifying a parameter that is present in the calculation of many nodes, it is recommended that you send the modification to a node from the Console, and then pass this modification to the other nodes through a FireWire link, namely from one calculation subsystem to another instead of sending to all the computing nodes from the console.

Two typical parameter modification blocks, which can be used to interact with the simulation from the Console are the Manual Switch and the Slider Gain:

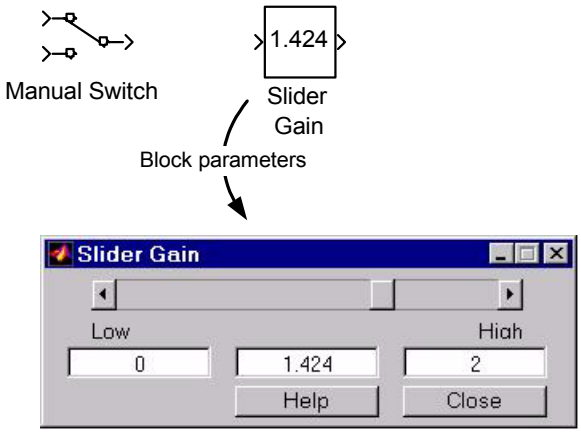


Figure 7-1 Slider gain.

The Manual Switch block allows discrete modification of the parameter; the Slider Gain block allows continuous modification over a specified range.

To modify the parameters of a filter, for example:

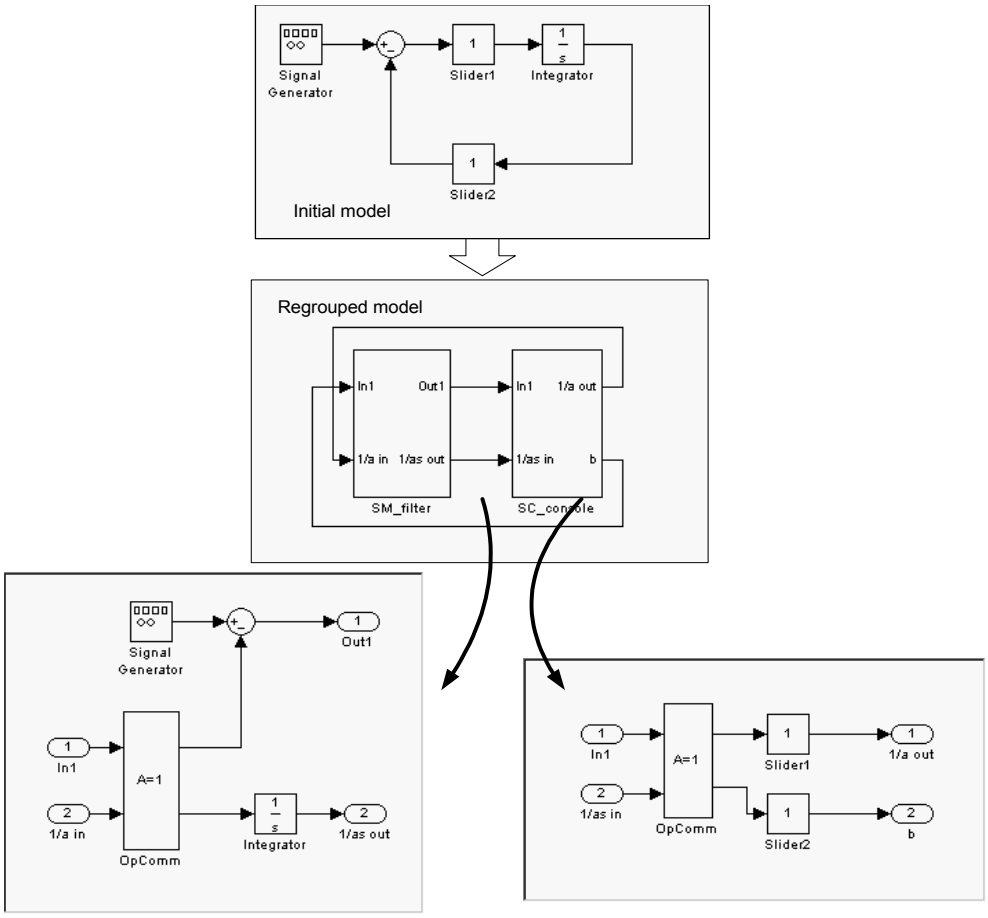


Figure 7-2 Modifying feedback gains using the Console.

8 Sample Simulation

A demonstration model is supplied with the system. This model has been designed to function properly in the RT-LAB environment, and allows you to review all the steps required to operate the system. Using the information provided in this Manual, you should be able to successfully open the demonstration file, group the model into its required subsystems, set its parameters, and run its simulation on your cluster.

1. Open the Demonstration file

To access the RT-LAB sample simulation's Demo file, you must have properly installed hardware and software for RT-LAB. For full installation requirements, see 4, Installation.

1. Open the demonstration file *rtdemo1_init* using your Simulink or SystemBuild block diagram editor.

   - For RT-LAB with MATLAB, this file is in the directory <*RT-LAB Installation dir*> / *Simulink* / *models*/ *rtdemo1*.

   For more information on opening files using the editor, refer to your Simulink documentation.

2. This model represents a second order system (a double integrator) that is controlled by a PID controller. You can select the reference position you want to system to get to and you can track the controlled response of the system on the display scope.

2. Group the Subsystems

The demonstration model consists of a global model divided into two subsystems, as is shown in Figure 8-1. Each subsystem is then distributed to a target node.

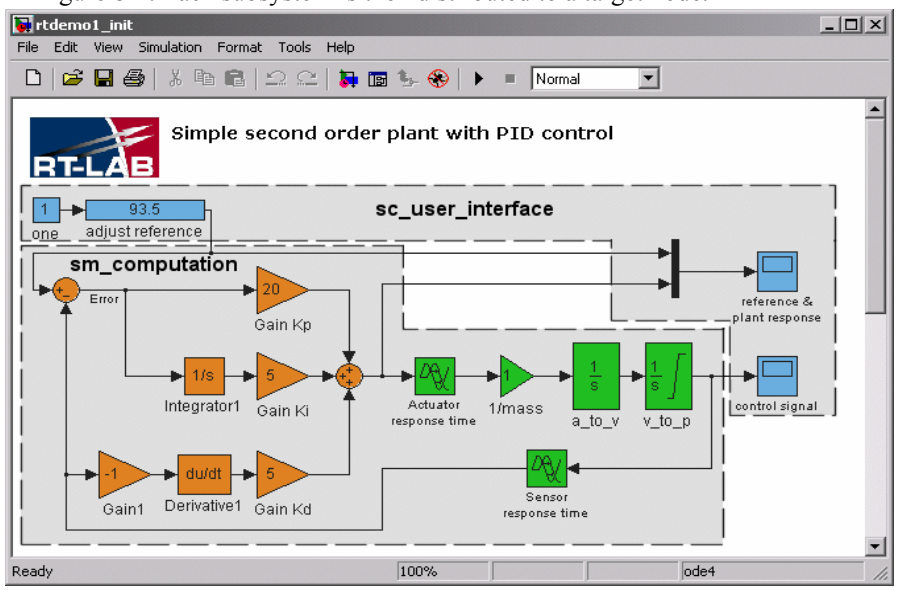


Figure 8-1 Grouping of an original model.

The final result of the grouping [is saved in the file *rtdemo1*, as shown in the following figure.

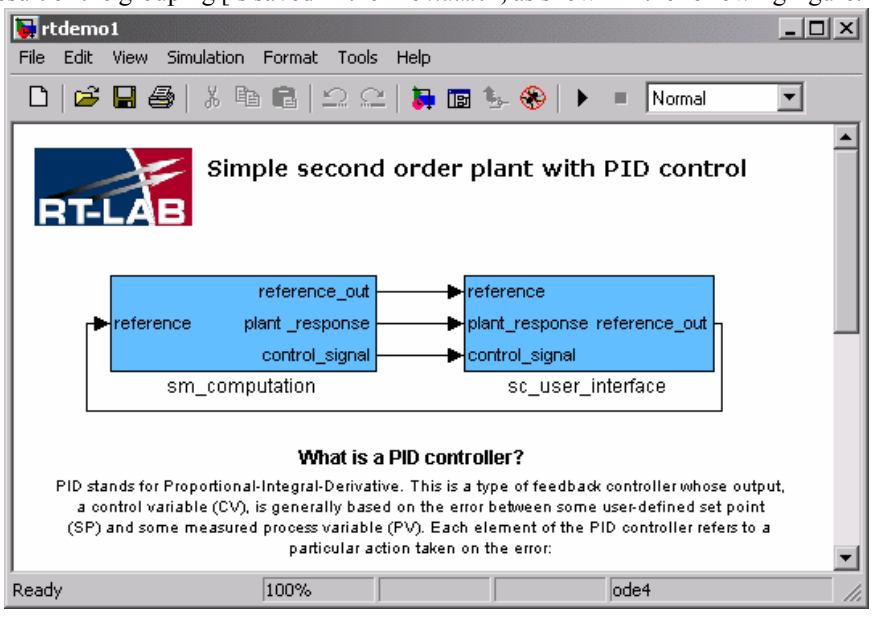

rtdemo1
File Edit View Simulation Format Tools Help
Normal
Simple second order plant with PID control
RT-LAB
reference
reference_out
plant _response
control_signal
sm_computation
reference
plant_response reference_out
control_signal
sc_user_interface
What is a PID controller?
PID stands for Proportional-Integral-Derivative. This is a type of feedback controller whose output, a control variable (CV), is generally based on the error between some user-defined set point (SP) and some measured process variable (PV). Each element of the PID controller refers to a particular action taken on the error:
Ready
100%
ode4

Figure 8-2 Final regrouped model.

For any given model, there must be only one master calculation subsystem, and its name must begin with the prefix SM_. There is also only one Console subsystem, and its name must begin with the prefix SC_. There can also be one or more Slave calculation subsystems, and their names begin with the prefix SS_. See 3, Building a Distributed Model for RT-LAB for a detailed description of each subsystem function.

In the case of this Demo file, the grouped model is composed of two subsystems:

- SC_user_interface includes the reference constant and slider gain, as well as the scope blocks for the reference and plant response and the control signal;
- SM_computation contains all the computational elements of the model which form the second-order plant and PID controller;

For full details on grouping your model into its subsystems and preparing the block diagram for use with RT-LAB, see 3, Building a Distributed Model for RT-LAB and 4, RT-LAB Blocks.

3. Start RT-LAB

1. Double-click on the *RT-LAB MainControl* program icon located on the desktop.

   This will start RT-LAB and open the Main Control Dialog box.

Note: The *MetaController* application must be running when using the RT-LAB software. If selection of the model didn't work or froze up the Dialog box, verify that the *MetaController* is running. If it is not, start the application from the RT-LAB program menu.

See Online help for more information about panels.

4. Select the Model

1. Click on the *Open Model* button in the Main Control Dialog box.
2. Double-click on the filename for the desired grouped model (for the purposes of this demonstration, choose *rtdemo1.mdl*).

   Once the model has been selected, its filename appears above the *Open Model* button.

5. Separate, Generate, and Compile the model

1. To change compilation parameters, right click on the *Compile* button. If no parameters changes are required, go to step 2.

   The following options are selected by default:

   - *Separate Model:* separation of the grouped model into sub-models;
   - *Generate Code*: generation of the C code for each sub-model;
   - *Compile Generated code*: code compilation for the various subsystems.

   The following option is available only if with RT-LAB for QNX target:

   - *Transfer files*: transfers all the appropriate files to your compilation node for compilation and processing.

   Diagnostics are displayed in the RT-LAB display panel, which appears on the screen.

2. Click on the *Compile* button on the Main Control dialog box.

   If this is the first time the model is being compiled on your system, a check box will appear that allows you to compile your model under different environments and with different parameters.

3. The message `Compilation finished ......` appears on the screen when the compilation process is completed.

Note: The user should carefully examine any messages displayed on the screen, to make sure the model separation, code generation, and compilation have been successfully accomplished.

6. Assign Nodes

At the end of the preceding step, an executable file is generated for each subsystem of the model.

1. Click on the *Assign Nodes* button to assign each computation subsystem to a physical target node.

   A dialog box will appear, allowing you to assign subsystems to the nodes in your cluster. In the sample Demo model provided, only *SM_Master* appears, since this is the only computation subsystem defined in the block diagram.

   If no nodes appear in your *Assign Nodes* column, see 3, Configuration to check your system configuration.

2. Double-click the physical node you want to use from the *Remaining Physical Nodes* list.

   This tells the system to run the subsystem *SM_Master* on the selected physical node. You can select another physical node configuration and execute the model again without having to recompile.

   See your QNX documentation for physically connecting a new node to your system. See 3, Configuration for details about adding a QNX target node to your RT-LAB configuration.

7. Load your model

1. Make sure the *Use model Console* option is selected on the Main Control Dialog box.

   This option automatically brings up the associated Console whenever a model is loaded.

2. Click on the *Load* button.

   This will load the files to be executed by the various nodes.The MATLAB program starts automatically, and the Console subsystem appears in the RT-LAB display windows, showing the simulation status and the communication diagnostics between each node in the system.

8. Run the Simulation

1. Click on the *Execute* button in the Main Control Dialog box, to start simulating the system.

2. Verify that the messages displayed in the RT-LAB display window at startup confirm that the model is in `Run` mode.

   This message is your confirmation that the model's simulation has started correctly.

For full details on compiling a model and executing a simulation, see 7, Running Your Simulation.

9. View your Data

At the end of the preceding step, you will have confirmed that the simulation is running properly.

1. Start the simulation by clicking on the *Play* button on the Simulink Console if it is not started.

   The Console is your link with the real-time simulation. The simulation output can be viewed through the scope provided in the SC_Console subsystem.

   - Double-click on the "adjust reference" block of the simulink console. You can adjust the reference value from 0 to 100. The first scope will display the reference signal as well as the system's actual position. The other scope will display the control signal computed by the PID controller..

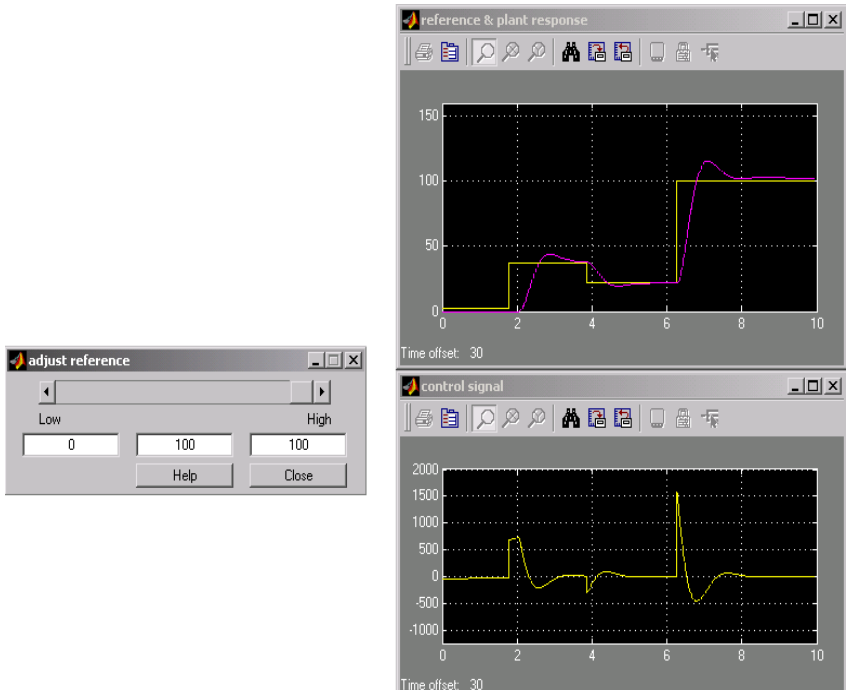


Figure 8-3 ***rt_demo1*** scope display.

For full details, see 5, Acquiring and Viewing Data.

10. Quit

To stop the simulation in order to make alterations to the current model:

1. Click on the *Reset* button in the Main Control Dialog box to stop the simulation's execution.
2. Click on *Edit* to modify the original block diagram.

To close the current model and open another one:

1. Click on the *Disconnect* button to disconnect the first model
2. Click on the *Open Model* button to open the next model.

To quit the program once a model has been reset:

1. Click on the *Close* button in the Main Control Dialog box.

A1 Setting up a Serial Connection to an RT-LAB Target

RT-LAB provides the option of setting up a serial link between the Command Station and any target node. This option is of benefit to users who, for example, may want to change network configuration from within the field, but whose target node does not include a monitor. It also provides alternate access (in addition to Ethernet) to the target, in case the Ethernet interface should malfunction or not be set up.

To configure your target node for serial connection:

Access the target through Ethernet or by using a monitor and a keyboard.

1. Open the target node's initialization file, /etc/config/sysinit.x, where `x` is the target's node number.

2. Change the last line in the file, which contains the `tinit` command, so that it reads:

   `tinit -T con* -t con1 -T ser1 &`

   In this case, `ser1` designates the COM1 serial communication port of the RT-LAB target node.

3. Type **`shutdown -f`** to reboot. (This will reset the target system.)

To set up a serial link:

Once the target is configured, physically connect the two nodes with a Laplink (DB9 or DB25 RS232 crossover) cable.

1. Log onto the target machine.

2. Launch a terminal program such as HyperTerminal.

   The Windows terminal program is found under the operating system's Start menu, at Start > Programs > Accessories > Hyperterminal path. While this is the program that is typically used, any similar terminal program can be used.

3. Specify the COM port that you are using (the one wher the cable is plugged in) and specify the following parameters, to be able to communicate with the QNX target PC:

   - 9600 bits per second
   - 8 data bits
   - no parity
   - 1 stop bit
   - hardware flow control.

A2 Coding an RT-LAB icon to Make I/O and Timer Icons

This Appendix describes the use of Simulink S-functions and SystemBuild UCBs with RT-LAB. The first section, *A2.1, Developing icons with RT-LAB*, outlines how S-functions and UCBs relate to RT-LAB, and describes the knowledge required to use them. Sections *A2.2, Simulink S-Functions* and *A2.3, SystemBuild UCBs*, briefly describe S-functions and UCBs, and section *A2.4, Creating icons with RT-LAB*, presents the RT-LAB features available to S-functions and UCBs. Finally, sections *A2.5, Programming an RT-LAB S-Function for an I/O icon* and *A2.6, Programming an RT-LAB S-Function for a Timer icon* present and explain two examples of S-function programming: the first for a simple I/O application and the second for a synchronisation/timer icon.

A2.1 Developing icons with RT-LAB

S-functions (Simulink) and User Code Blocks (SystemBuild), also called UCBs, are computer language descriptions of icon behavior. These are the tools that allow users to create icons for use in models. Just as icons can be made to simulate special dynamics, they can also be used to access I/O cards. This document explains how to develop I/O and timer S-functions or UCBs using features specific to RT-LAB. The information presented here is also generic enough to help developing any kind of basic S-functions and UCBs.

RT-LAB is designed to work with any S-function or UCB that is compatible with Simulink, Real-Time Workshop and SystemBuild. However, to maximize RT-LAB's full potential, it is possible to add some RT-LAB features to a standard S-function or UCB.

The RT-LAB feature will allow you to:

- Execute an I/O-triggering/accessing routine at the beginning of the model step (CB_ExecuteIO)
- Execute a routine when the model is paused (CB_Pause)
- Execute a routine when the model is reset (CB_Reset)
- Wait for the timer-interrupt in the case of a timer icon (CB_Wait_for_timer)
- Execute one or more initialisation routines (CB_Init, available as of RT-LAB 4.2)
- Execute one or more termination routines (CB_Close, available as of RT-LAB 4.2)

A2.1.1 Required Knowledge

It is expected that users are familiar with C language. S-functions and UCBs are introduced here but detailed instructions are provided with Simulink and SystemBuild software:

For S-functions:

<MATLAB_install_dir>\help\pdf_doc\simulink\sfunctions.pdf

<MATLAB_install_dir>\simulink\src\sfuntmpl.doc

For UCBs:

<MATRIXX_install_dir>\help\help.html

A2.2 Simulink S-Functions

An S-function can be visualized as a source code description of an icon's behavior. Having programmed this file, all there is to do is to compile it as a dll (dynamic link library) and to associate it with a standard S-function block in order to use it. The developer has thus created an icon that has all the characteristics of usual Simulink blocks.

An S-function is not a singular function, but rather a collection of functions that can be called by Simulink while your model is executing. These functions can be sorted in three different categories:

- initialization functions: used, for example, to set number of icon input/outputs and to verify for errors in the icon's parameters.
- task functions: these are called during model execution and perform the icon's main operation, such as reading its inputs and calculating its outputs
- termination functions: executed when the model is stopped and used, for example, to free allocated memory

All of these functions are written in C language and are located in a single source file. Each of these functions receive as parameter a pointer to a *SimStruct* structure that contains information to be made available to all of these functions. A *SimStruc* is allocated by Simulink for each instance of an S-function related icon, allowing them to be fully independent from each other (i.e. each one of them having its own memory space). This *SimStruc* also contains a pointer that can be used by the user to point to a memory space assigned to a structure, called LocalData, containing user defined variables that can thus be made available to all member functions of the S-function. This LocalData structure can be any size and must be allocated by the user in one of the initialization functions.

One must be careful while using global variables S-function files since such a variable will be allocated only once in a given model for all instances of a given S-function. Such a problem is detoured by the use of the LocalData structure.

A2.2.1 Main S-Function Routines

The following overview outlines Simulink's main S-function routines. The information presented here is just an overview. Every topic will be explored deeper as examples will be explained in the following sections.

A1.2.2.2 mdlInitializeSizes

Simulink calls this routine at the start of simulation to set the size of the icon input/output ports and set the number of icon parameters such as the gain to be applied to the input of the S-function. Simulink also calls this routine while it edits the model, to determine the number of input and output ports.

A1.2.2.3 mdlInitializeSampleTimes

Simulink calls this routine to set the sample time(s) of the S-function.

A1.2.2.4 mdlInitializeConditions

Simulink calls this routine before the model executes to initialize the icon. In the case of an icon that is writing to a file, the file will be opened. This function is the best place to initialize variables and allocate memory prior execution of the S-function's main body. This is also where the memory for the LocalData structure is allocated.

A1.2.2.5 mdlOutputs

Simulink calls this routine during simulation, each time the S-functions' outputs need to be updated. This is usually where the main task of the S-function is programmed since this function is called every calculation step.

A1.2.2.6 mdlTerminate

Simulink calls this routine to perform tasks at end of the simulation. If the S-function has no termination tasks to perform, this routine will be empty. In the case of an icon that is writing to a file icon, the file will be closed in this function.

The following figure shows the execution of an icon during a model execution:

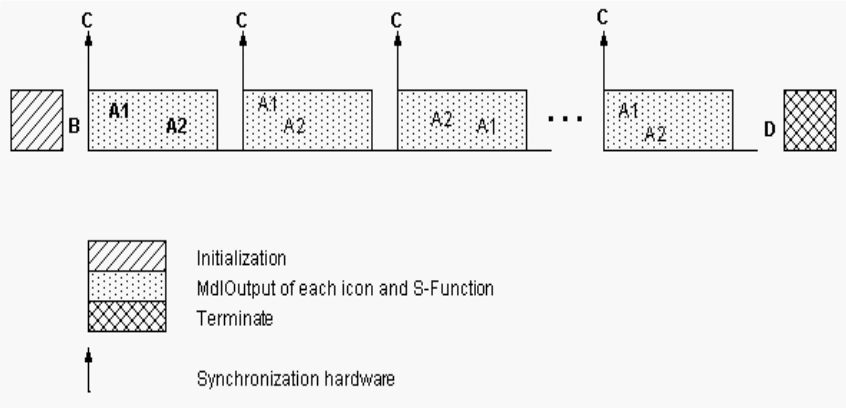


Figure A2-1 Execution of a Simulink model

The Initialisation function "B" is called once at the beginning, then the mdl_output functions, A1 and A2, are called at each step of the execution. Point C corresponds to the

synchronisation interrupt that indicate the beginning of a step. Finally, the Terminate function, D, is called when the model is reset.

As one can notice on the figure, the mdl_output functions A1 and A2 are not always regularly spaced in time. This is due to several factors, a major one being operating system activity. This may be a drawback when performing IO tasks but, as it will be explained later, such a problem can be detoured with RT-LAB.

A2.2.7 How to create an S-Function

Here is a quick step-by-step description of the procedure to use an S-function in a Simulink or RT-LAB context.

1. Write the S-function C source file, using one of the previously mentioned templates.
2. Compile this file as a dll using any C compiler or Matlab. To use Matlab, put this source file in one of Matlab's file paths and simply use the command: *mex filename.c*.
3. Take a Simulink S-function block and put it in a model.
4. Double-click on it and put the dll's name (without extension) in the first field.
5. Also add icon parameters in the second field. These can be either numerical values, variables or strings. Separate each parameter by a comma.

All is now ready for use in a Simulink environment. The following steps are necessary for an RT-LAB application.

1. Click on the Configuration button from the MainControl panel.
2. Press the Advanced button.
3. Go to the Files & Command tab.
4. Add your C source filed in the extra files section (use the Add and Browser button), along with all the header files (*.h) required.

A2.3 SystemBuild UCBs

A UCB can be visualized as a source code description of an icon's behavior. Having programmed this file, the only step left is to compile it as an object file and to associate it with a standard UCB block in order to use it. The developer has thus created an icon that has all the characteristics of usual UCB blocks.

A UCB a singular function that is called by SystemBuild while executing the model. Tests are made on a variable to know the phase of execution of a model, whether it is initialization or main execution.

This function receives a long list of parameters among which are, number inputs, outputs and parameters A pointer is also received in order to be used for accessing user allocated memory space for UCB specific variable. These variables are grouped in a structure (that

we will refer to as LocalData) allocated by the user in the initialization. This allows all instances of an UCB to be fully independent from each other..

One must be careful while using global variables in UCB files since such a variable will be allocated only once in a given model for all instances of a given UCB. Such a problem is detoured by the use of the LocalData structure.

A2.4 Creating icons with RT-LAB

Most of the features that RT-LAB adds to S-functions and UCBs are intended for I/O and timer applications. They allow, among other things, to ensure that I/Os will be accessed at regular intervals and to set I/Os before simulation is paused or reseted

With a standard Simulink S-function or SystemBuild UCB, the I/O will be accessed in the corresponding output section. The result is that you are not ensured that the delay between two calls to this function will be constant, as displayed with mdl_output functions A1 and A2 on figure X.1. There is no way to know whether the model is paused, or to include multiple initialisation or termination functions. In the case of a digital output icon used to generate a square wave, inconsistent delays may cause a variation in the signal frequency.

The RT-LAB function solves these problems by allowing the user to register callback functions with some events. For instance, by using the ExecuteIO callback, a routine can be executed at the beginning at each step to ensure constant intervals between the routine's calls. It is also possible to register a Pause routine that will allow the user to set the I/O output to secure values.

A2.4.1 RT-LAB Routines

The following routines can be used to register a callback function. For complete information, please refer to the OpalIO.h file:

<RT-LAB install dir>/common/include_target)

A1.2.4.2 Register_CB_Init

This routine, allows users to register a function that will be called during the model's initialization phase. The registered function can be used to allocate the memory required by the model, and initialise the I/O board.

A1.2.4.3 Register_CB_Close

This routine, allows users to register a function that will be called when the model is reset. The registered function can be used to de-allocate memory used by the icon, and set the I/O to a secure Reset value.

A1.2.4.4 Register_CB_ExecuteIO

This routine allows users to register a function that will be called during the model's ExecuteIO event. The registered function is called just after the Timer Interrupt, which indicates that it is time to start a new computation step. This ensures that the registered routine will be called at regular intervals at the beginning of each calculation step.

Note: The functions registered for the ExecuteIO event will never be called if the model is running is simulation (non-synchronised) mode. Therefore, when using the ExecuteIO function, the user can be assured that there will not be any problem if a user runs in Simulation mode by mistake.

A1.2.4.5 Register_CB_PauseIO

This routine allows users to register a function that will be called during the model's Pause event, and is called whenever the simulation model is paused.

A2.5 Programming an RT-LAB S-Function for an I/O icon

An S-Function template is supplied with RT-LAB:

<RT-LAB install dir> / doc / Simulink / S-function / template / sfuniotmpl.c

While designed as an example of parallel port writing and PC speaker control, the template can be used for other purposes like Analog Input or Digital IO. This section discusses the RT-LAB S-function code as it relates to using I/O cards, registered function by registered function, including:

- the Template Header, which defines headers and structures;
- InitializeConditions, which starts reads icon parameters and initialises variables and IO card;
- MdlOutputs, which reads and writes icon input/outputs and accesses IO card;
- MdlTerminate, which resets the acquisition card;
- MyExecuteIO, which sets the I/O board's output value should synchronisation be requested; and
- MyFunctionPause, which sets a secure pause value whenever the model is paused.

Note: Note that the code segments presented here have been shortened for clarity purpose. For the complete code refer to the abode file.

The code in the template has been prepared to run under QNX but aside from operating system specific functions and features (like IO port addressing and access), every information presented here is also valid for any RT-LAB supported system.

A2.5.1 Template Header

This first part of the template defines the headers declarations, and LocalData structure, which is the structure that will contain all values that your icon needs. It can be any size and is fully customisable for any given application. One useful value to include in the structure could be the card address, as shown below:

```
#if defined(RT)
#include « OpalIO.h »
#include « OpalError.h »
#endif

#define NB_PARAM2
#define NBINPUT1
#define NBOUTPUT0



/* Structure used in MyFunction_… call */
Typedef struct
{
Intcard_address;
InputRealPtrsType input;
Doubledata_output;
IntSynchro;
} LocalData;

#if defined(RT)
static void MyFunction_ExecuteIO(void *GenStruct);
static void MyFunction_Pause(void *GenStruct);
#endif
```

One can also notice that the numbers of S-function parameters, inputs and outputs have been defined. Using these definitions allows the programmer to leave the mdlInitialiseSizes function, as programmed in the example, untouched. This last function will not be discussed further in this document. Also added are all user defined function headers.

A1.2.5.2 InitializeConditions

The mdlInitializeConditions() function reads icon parameters from the icon mask and uses them to initialise variables. While programming an IO icon, card initialisation should occur in this function. Any call-back function used should be registered in mdlInitializeConditions, as with the Register_CB_ExecuteIO and Register_CB_Pause functions in the example below. A test is made on the Synchro variable to find out if card related operations are to be executed in the MyFunction_ExecuteIO call-back function or in the mdlOutput function.

Note: In the case of an analog output card, mdlInitializeConditions is also where the I/O card's voltage range is set.

```
static void mdlInitializeConditions(SimStruct *S)
  {
#if defined(RT)
LocalData *ld=LD_PTR;/* Pointer to local data structure */

/* Get the acquisition card address (parameter 1)*/
ld->card_address= (int)mxGetPr(ssGetSFcnParam(S,0))[0];

/* Get input port pointer */
ld->input = ssGetInputPortRealSignalPtrs(S,0);

/*Initialize card */
if(!MyCardInit(ld->card_address))
ERROR_STATUS(INIT_IO,S_FUNCTION_NAME)
else
{
/* Register MyFunction_ExecuteIO for the ExecuteIO event
** This function will be called using ld, after
OpalWaitforTimer */
if(ld->Synchro)
Register_CB_ExecuteIO(ld,MyFunction_ExecuteIO);
/* Register MyFunction_Pause for the Pause event.
```

```
** This function will be called using ld, when the model is
paused */

Register_CB_Pause(ld,MyFunction_Pause);

}


#endif

}
```

A1.2.5.3 MdlOutputs

The mdlOutputs() function is called every calculation step and performs the icons'main task. The example here reads the icon input and sends it to the user defined MyCardOutput function. This function could perform any given task with this information like, for example, writing this data on a given channel of DAC card. Notice that a test is made to find out if card related operation is to be executed here or if it had been done in the MyFunction_ExecuteIO function.

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
#if defined(RT)
LocalData *ld = LD_PTR;
if(!(ld->Synchro))
MyCardOutput(ld->card_address,*ld->input[0]);
#endif
}
```

A1.2.5.4 mdl_terminate

The mdlTerminate() function will reset the acquisition card, and set the output at a given reset value.

Note: This final value must be set at a safe value for any hardware that could be connected to your I/O board.

```
static void mdlTerminate(SimStruct *S)

{

#if defined(RT)

LocalData *ld = LD_PTR;

nosound();        // Stop sound


outp(ld->card_address,RESET_VALUE);
```

```
 #endif


}
```

A1.2.5.5 MyExecuteIO

The MyFunction_ExecuteIO() function was registered in mdlInitializeConditions() using Register_CB_ExecuteIO(), and will therefore be called at each beginning of step. Access the I/O board through this function to set its output value.

Note: The value sent to the card is written into the LocalData structure by the mdlOutput() function. This MyExecute function simply sends the value to the I/O card. Most calculations should be avoided in call-back functions since many ExecuteIO call-back function can be register and are also waiting to be executed quickly at the beginning of each calculation steps.

```
static void MyFunction_ExecuteIO(void *GenStruct)

{


LocalData *ld = (LocalData*)GenStruct;


MyCardOutput(ld->card_address, * ld->input[0]);


}
```

A1.2.5.6 MyFunctionPause

The MyFunction_PauseIO() function was registered in mdlInitializeConditions() using Register_CB_PAUSEIO(). This function is called when the model is paused, and sets the output value of the I/O board to a secure pause value.

```
static void MyFunction_Pause(void *GenStruct)

{

LocalData *ld = (LocalData*)GenStruct;

nosound();


MyCardOutput(ld->card_address,PAUSE_VALUE);
```

```
}
```

A2.6 Programming an RT-LAB S-Function for a Timer icon

A timer icon uses a hardware timer to synchronize the execution of a model. The main purpose of the RT-LAB Timer S-Function is to wait for the synchronization interrupt to occur.

Synchronization is usually done using icons like OpSyncDccXp. When using your own hardware timer card to synchronize IO, you'll have to create a timer icon as described in this section. The template supplied with RT-LAB can be used to develop RT-LAB timer icons:

<RT-LAB install dir> / doc / Simulink / S-function / template / sfuntimertmpl.c

The template is also an example that uses the QNX system clock instead of a card in guise of timer. This section discusses the RT-LAB S-Function code as it relates to timer icons, registered function by registered function, including:

- the Template Header;
- InitializeConditions;
- MdlOutputs; and
- MdlTerminate — as with programming an RT-LAB S-Function for an I/O icon — and
- MyFunction_WaitForTimer, which blocks the simulation until the timer interrupt is received; and
- MyFunction_DisableTimer, which disables the timer interrupt.

The basic idea of a synchronisation icon is to wait for a signal that will launch the beginning of a new calculation step. This signal usually comes in the form of a timer card producing an hardware interrupt (IRQ). The system clock can also be used to produce this interrupt, like in the template example.

The code in the template has been prepared to run under QNX but aside from operating system specific functions and features (ISRs, timer access and control), every information presented here is also valid for any RT-LAB supported system.

A2.6.1 Template Header

This first part of the template defines the header declarations, and LocalData structure, which is the structure that will contain all values that your icon needs. One useful value to include in the structure could be the step size, as shown below:

```
#if defined(RT)
#include « OpalIO.h »
#include « OpalError.h »
#include <sys/proxy.h>
#include <sys/irqinfo.h>
#include <sys/kernel.h>
#endif


/* Structure used in MyFunction_… call */
Typedef struct
{
double              old_StepSize;
double              actual_StepSize;
int                 FirstTime;
int                 InterruptNb;
} LocalData;

#if defined(RT)
volatile unsigned   counter;
pid_t               proxy;
static int          iid;
static int          Paused;

#                   pragma off( check_stack );
pid_t far Myhandler()
{
if(Paused)
                    return (0);
else
{
                    counter++;
```

```
                    return (proxy); // return proxy
}
}
#                   pragma on( check_stack );

static void MyFunction_DisableTimer(void *GenStruct);
static void MyFunction_WaitForTimer(void *GenStruct);
#endif // if defined RT
```

Also defined here is the interrupt service routine (ISR) that will respond on the synchronisation interrupt and signal the MyFunction_WaitFortimer routine to stop waiting.

A1.2.6.2 InitializeConditions

The mdlInitializeConditions() function uses the input parameters to start the acquisition card, and fills out the LocalData structure. The timer card input clock and count configuration are set using mdlInitializeConditions. Any callback function used should be registered in mdlInitializeConditions, as with the Register_CB_SingleStep, Register_CB_WaitForTimer and Register_CB_Pause functions in the example below.

Note: Since a model can only have one synchronization source, only one function can be registered with Register_CB_WaitForTimer().

```
static void mdlInitializeConditions(SimStruct *S)
{
#if defined(RT)

LocalData *ld = LD_PTR;/* Pointer to data structure */



ld->actual_StepSize = OpalGetCalculationStep();
ld->old_StepSize = 0;
```

```
Paused =1;
if ( !MyInitTimer() )
{
OpalPrint("Unable initialize the timer\n");
ERROR_STATUS(INIT_IO,S_FUNCTION_NAME);
}
/* Get a proxy for the interrupt handler to kick */
if( ( proxy = qnx_proxy_attach( 0, 0, 0, -1 ) ) == -1 )
{
OpalPrint("sfunTimertmpl: Unable to attach proxy. \n");
ERROR_STATUS(INIT_IO,S_FUNCTION_NAME);
}
ld->FirstTime = TRUE;

/* Attach to the interrupt */
if(iid=qnx_hint_attach(INTERRUPT_NB,&Myhandler,
FP_SEG(&counter)))
{
OpalPrint("Unable to attach interrupt handler. \n");
ERROR_STATUS(INIT_IO,S_FUNCTION_NAME);
}
ld->interruptNb=INTERRUPT_NB;

Register_CB_WaitForTimer(ld,MyFunction_WaitForTimer);
Register_CB_Pause(ld,MyFunction_DisableTimer);
 Register_CB_SingleStep(ld,MyFunction_DisableTimer);

#endif /* RT */
  }
#endif /* MDL_INITIALIZE_CONDITIONS */
```

A1.2.6.3 mdl_outputs

The mdlOutputs() function will check for any change in the calculating step, and automatically update the timer speed as required.

Note: A timer icon MUST perform this task in the mdlOutput function, since RT-LAB supports time step changes after the model is started.

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
…
#if defined(RT)
LocalData*ld=LD_PTR; /* Pointer to LocalData structure */

/* Get calculation step */
Paused=0;
ld->actual_StepSize = OpalGetCalculationStep();

/* Compare with last calculation step */
if(ld->old_StepSize != ld->actual_StepSize)
{
MyChangeTimerSpeed(ld->actual_StepSize);
OpalPrint("Speedchanged:%.4f\n",ld->actual_StepSize);
}

ld->old_StepSize = ld->actual_StepSize;

#endif
…
}
```

A1.2.6.4 mdl_Terminate

The mdlTerminate() function is called when the simulation is reset, and will reset the acquisition card and set the output of the reset value. [**Hint:** Disable the interrupt when using this function, so that your PC will not have to handle interrupts after the model is stopped.]

Note: This final value must be set at a safe value for any hardware that could be connected to your I/O board.

```
static void mdlTerminate(SimStruct *S)

{

…

#if defined(RT)

Paused=1;

qnx_hint_detach( iid );

#endif

…

}
```

A1.2.6.5 WaitForTimer

The MyFunction_WaitForTimer() function was registered in mdlInitializeConditions() using Register_CB_WaitForTimer. This function will block until the timer interrupt is received by the timer card. Since the timer is set according to the model's step time value, this will synchronize the model's step execution.

Note: The WaitForTimer function uses QNX blocking functions like Receive() and Creceive(). Polling with a "for" or a "while" loop will increase the CPU work charge and prevent lower priority tasks (like Data Acquisition) from being done while the function waits for the interrupt.

```
#if defined(RT)


static void MyFunction_WaitForTimer(void *GenStruct)

{

LocalData *ld=(LocalData*)GenStruct;


Paused=0;
```

```
Receive(proxy,0,0);

}

#endif
```

A1.2.6.6 DisableTimer

The MyFunction_DisableTimer() function was registered in mdlInitializeConditions() using Register_CB_Pause() and Register_CB_SingleStep(). It disables the interrupts so the PC does not accumulate interrupts while in Pause.

```
#if defined(RT)
static void MyFunction_DisableTimer(void *GenStruct)
{

      LocalData *ld = (LocalData*)GenStruct;

      Paused=TRUE;
      ld->FirstTime = TRUE;
}
#endif
```

Glossary

Acquisition Board	See Data Acquisition Board.
Acquisition Frames	See Frames.
A/D Converter	A device that converts data from analog to digital form. See also D/A Converter.
Aliasing	The effect of sampling a signal, resulting in additional harmonics in the signal. This effect is increased when the sampling rate is insufficiently fast for the signal being sampled. See also Anti-aliasing filter.
Analog	Analog systems handle information affected by continuous change and flow, like voltage or current. In contrast, digital data is represented in discrete units (the binary digits 1 and 0). See also Digital, A/D converter, and D/A converter.
Anti-aliasing filter	A low-pass filter that counteracts the effects of aliasing.
Base Address	A memory address that serves as the starting point for programmable registers.
Block	Graphical unit that makes up part of a block diagram, representing a specific operation, calculation, or subsystem within the system represented by the block diagram. See also Block Diagram, Block Library, and Communication Block.
Block Diagram	A diagram of a dynamic system, in which the principal parts are represented by blocks that show both the basic functions of the parts, and the data flow connections between them.
Block Library	A collection of reusable blocks. Library blocks are dragged from the Block Library and dropped into a model's block diagram.
Buffer	A reserved data storage area used in RT-LAB to compensate for the difference in speed when transferring data from one device to another.
Bus	A computer's expansion connector, into which boards are inserted, and through which all communication between the computer and your board occurs. There are several different expansion buses available, including the ISA, PCI, PC/104PC/104, IP-Module buses.
Calculation Node	See Target Node.
Calculation Step	See Step Size.

Carrier Board	Computer board that can be reconfigured by inserting IP modules into its sockets.
cLAN	Giganet Inc.'s server cluster Local Area Network. This 1000Mb/s network uses a cable to link computers into a cluster, and bypasses operating system interfaces to provide a direct path for applications.
Client	In a Client/Server computer architecture, the client makes requests to the server
Cluster	Two or more computers linked together to function as one system. Clustered computers usually have specialized hardware to connect them. Compare with Network.
Command Station	Known in former versions of RT-LAB as the *Host*. The computer that serves as the user interface, housing RT-LAB's control panels, and used to design, control, and analyze simulations.
Commercial-off-the-shelf (COTS)	Readily available commercially.
Communication Block	Also known as an *OpComm block*, this is a simple feed-through block that receives all signals coming into the subsystem into which it is placed. Communication blocks provide information to RT-LAB about the type and size of signals being sent between subsystems. They are also designed such that the behavior of the system is the same when simulated on the Command Station, or when distributed across target nodes.
Communication Rate	Rate at which data is exchanged between nodes.
Compilation Node	Known in former versions of RT-LAB as the *Development Node*. The computer in RT-LAB that compiles C code, loads the code into each target node, and debugs source code.
Computation Node	See Target Node.
Console	Loaded onto the Host, the Console is the command terminal program that serves as the user interface, and communicates with the target nodes. The Console subsystem (SC_) contains all element blocks related to viewing data and sending commands.
COTS	See Commercial-off-the-shelf.
D/A Converter	A device that converts data from digital to analog form. See also A/D Converter.
Data Acquisition Board	Computer board dedicated to reading data from external components.

Decimation Factor	When a signal is decimated by a factor *N*, every *n*th sample of that signal will be acquired.
Development Node	See Compilation Node.
Digital	Digital data is represented in discrete units comprised of the binary digits 1 and 0.See also Analog, A/D Converter, and D/A Converter.
Distributed	Separated into parts or units. A distributed model is one whose calculations have been divided among a number of subsystems that are represented by blocks. Distributed computing involves dividing computations across a cluster of processors. A distributed simulation is one done using distributed computing.
Dual-processor	See Multiprocessing.
Embedded System	A system where hardware and software combine to form a component of a larger machine.
FIFO	A First-In/First-Out memory buffer, where the first data received and stored is the first data sent out of the buffer.
FireWire	The informal name for High Performance Serial Bus (IEEE 1394). The High Performance Serial Bus can connect up to 64 devices, and transmit data at up to 400 megabits per second.
Fixed Step Size	See Step Size.
Frame	A group of consecutive data signals.
Giganet	See cLAN.
GUI	Graphical User Interface.
Hard-Real-Time	A system is hard-real-time when it guarantees time constraints will always be met within a given margin.
Hardware-in-the-Loop (HIL)	Configuration where external hardware components are physically integrated into the simulation system.
Host	See Command Station.
Host Name	For the purposes of hardware installation for an RT-LAB cluster, this unique name (translated into an IP address by a domain name server) identifies each physical node on the network.
I/O	Input/Output. See also I/O Address and I/O Board.
I/O Address	A unique address given to a peripheral I/O board.

I/O Board	Computer board dedicated to exchanging data between a computer and external components. I/O boards include AD and DA converters, I/O binary ports, I/O modules, carrier boards, com cards, quadrature decoders, etc.
I/O Card	See I/O Board.
I/O Device	See I/O Board.
I/O Module	See I/O Board.
IP Address	Internet Protocol address, shown in numeric form (123.231.32.2). See also Host Name.
IP Module	Industry Pack module. A small "piggy-back" I/O board that connects to ISA, PCI, or PC/104 carrier boards, and is used to configure them.
Master	The Master is responsible for the overall synchronization of the distributed simulation. The SM_ subsystem contains high-rate elements blocks that represent operations to be performed on signals and/or I/O icons. There is one Master per distributed simulation.
Multiplexing	Transmitting two or more distinct signals simultaneously. In block diagrams designed for use with RT-LAB, this is done using a *Multiplexing Icon*, also known as a *Multiplexor* or *Mux*, which allows multiple channels to be used, so that numerous signals can be sent through one module. The Multiplexing icon must be inserted at a block's input, and signals will be sent to the output in the same order (top to bottom) as they appear in the Multiplexing icon. Multiplexed signals must pass through a *De-multiplexing icon* at the block's output.
Multiprocessing	Running multiple processes (programs) in pseudo-parallel on a single-processor machine, or in parallel on a multi-processor machine (one using two or more processors in the same computer). See also Cluster and Network.
Multirate System	System that uses multiple step sizes for a simulation. A multirate system usually consists of a base step size and a set of step sizes that are integral multiples (integer dividends) of the base step size.
Multitasking	Running multiple processes on one or more processors.
Multithreading	Running multiple calculation threads per process. See also Thread and Single-Threading.
Network	Two or more computers linked together, and the hardware and software used to connect them. Compare with Cluster.
Node	See Compilation Node and Target Node.
Non-Real-Time	A non-real-time simulation is one that runs either faster or slower than real-time, depending on the complexity of the model and the speed of the processor.

NT Target	Target node whose operating system is Windows NT or Windows 2000.
Offline Simulation	Simulation executed in a non-real-time environment, like Simulink or SystemBuild, without using RT-LAB.
OpComm Block	See Communication Block.
Parallel	See Distributed.
Peripheral Device	See I/O Board.
Ping (Packet Internet Gopher)	Way to test or time the response of a TCP/IP connection. A PING sends a request to a node and waits for a reply. When you PING an address, you get a response telling you the number of seconds it took to make the connection.
QNX Compilation Node	See Compilation Node.
QNX Target	Target node whose operating system is QNX.
Quad-processor	See Multiprocessing.
Real-Time	A real-time simulation simulates events in the time-frame in which they would naturally happen. A real-time system is one that guarantees that calculations will be performed within specified time constraints, usually measured in milliseconds or microseconds. See hard-real-time and non-real-time.
Sampling Period	See Sampling Time.
Sampling Rate	The frequency, usually uniform, at which a signal is sampled (the number of analog-to-digital conversions per second). This term also indicates the rate at which a discrete-time system executes. In a multirate model, the base sampling rate is the step size.
Sampling Time	Known in previous versions of RT-LAB as the *sampling period*. The time period between samples (the number of seconds per analog-to-digital conversion). See sampling rate.
SC_	Prefix added to the name of the subsystem related to the Console.
Server	In a Client/Server computer architecture, the server answers requests from one or more clients.
Single-threading	Processing one transaction at a time. See also Thread and Multithreading.
Slave	Slave computers perform calculations and are synchronized by the Master. The Slave subsystems (SS_) contain operations to be performed on signals, and all elements other than display elements (which are found in the SC_ subsystem) and high-rate elements (which are found in the SM_ subsystem).

SM_	Prefix added to the name of the subsystem related to the Master.
Snapshot	A Snapshot is a record of a simulation's values, that can be taken at any point during the simulation. RT-LAB's Snapshot feature also allows for generating a report of these values.
SS_	Prefix added to the name of the subsystems related to the Slaves.
State Variable	In RT-LAB, a state variable is an independent variable that, within any given calculation step, depends only on previously generated inputs to calculate its value. This allows the variable to be calculated at the beginning of the step, while other computations continue.
Step Rate	Number of calculation steps per second.
Step Size	The time interval between consecutive calculation steps (seconds per calculation step). A model with a variable step size can modify its step size during its simulation; a model with a fixed step size will use the same step size value throughout its simulation.
Subsystem	A system that is part of a larger system. In RT-LAB, a model's block diagram is distributed into numerous subsystems to allow synchronization and distributed computing. See also SC_, SM_, and SS_.
Target Node	Known in former versions of RT-LAB as *Computation Node*, *Calculation Node*, or *Processing Node*. Real-time processing and communication computer on which the simulation's calculations are performed, that supervises the execution of a simulation, and communicates with other nodes.
Target Platform	The operating system platform of the target node.
TCP/IP	Transmission Control Protocol/Internet Protocol.
Thread	A sequence of instructions within a single process (program). See also Multithreading and Single-Threading.
Trigger	A discrete event used to initiate operations.

Index

A

A/D converters 25, 28
Acquisition
  Console 38
  data loss 34
  frames 37
  groups 33
    OpComm block rules for . . . 17
  parameters 33
  synchronous 37
  triggering 33
Algebraic loop 17
Aliasing 37
Analog
  input 28
  output 29
Analog-to-digital converters, see A/D converters

B

Base sampling rate 41
Block diagram
  inserting icons 25
  inserting OpComms 16
  using 6
Block Library 10, 25
Buffer 34

C

C/C++ 7
Calculation speed 10
Cards
  IP modules 26
  ISA 27
  named in icons 28
  PC/104 27
  PCI 27
Carrier board 26
cLAN 7
Code
  AutoCode 47
  C code 7, 47
  debug source code 5
  generating 47
  ISI C code generator 47
  loading 5
Command Station 4
Communication
  blocks 48
  cLAN 7
  delay 48
  Ethernet 5, 7
  non-real-time 17
  parallel 48
  rate 42
  real-time 5, 7, 17
  shared memory 7
  TCP/IP 7
Compilation Node 5
Compiling 41, 47
Configurations
  components 4
  hardware-in-the-loop 6
Console 7, 9, 13, 49, 53
  as graphic interface 7
  blocks 13
  communicate with target nodes . . . 34
  crash 37
  Load/Reset 54

D

D/A converters 25, 29
Data
  "holes" in display 35
  loss 34
  retrieval 13
  transfering 7
  viewing 38
Decimation factor 36
Demonstration file 51
De-multiplexing icon 28, 30
Digital
  input 30
  output 31
Digital-to-analog converters, see D/A converters
Drivers
  custom 12

E

Ethernet 5, 7, 34
Executing a simulation 55

F

Files
  .mat 34
  rtdemo1 51
FireWire 7, 11, 49
Frame 35, 37

G

Generating code 47
Graphic interface
  using the Console 7

H

Hardware
  I/O 45
  synchronization 12
Hardware-in-the-Loop 6, 13, 48
HIL, see Hardware-in-the-Loop

I

Icons
  adding 25
  block library 10, 25
  OpAnalogIn 28
  OpAnalogOut 29
  OpComm Blocks 32
  OpDigitalIn 30
  OpDigitalOut 31
  OpQuad-Decoder 31
  OpSync_VME 32
  OpSynch 40
  OpWriteFile 34
  setting parameters 25
Input/Output
  hardware synchronization 12
  I/O boards 5
  I/O communication 13
  I/O devices 6
  I/O modules 8, 25
  I/O Synchronize option 27
  inserting icons, see Icons
  multirate 45
  sample and hold 48
IP
  carrier boards 25, 26
  modules 25, 26
ISA cards 27

L

Libraries
  RT-LAB I/O icons 26
Loading 41

M

Main Control Panel
  Compile button 47
  dialog box 53
Manual Switch 49
Master 9, 13
  blocks 13
.mat file 34
Matlab
  Real-Time Workshop (RTW) 47
Minimum step size 11
Model
  compiling 41, 47, 53
  designing and validating 6
  dividing 14
  executing simulation 48
  generating 53
  grouping into subsystems 13
  interacting with 49
  loading 41, 48, 54
  modifying parameters 7
  multiple sample rates 40
  offline simulation 48
  performance parameters 49
  separating 53
  separation errors 44
  using RTW 40
Multiple sample rates 40
Multiplexing icon 29
Multiprocessing 7
Multirate mode
  and I/Os 45
  and SC_ 45
Multithread mode 40

N

Nodes
  assigning 54
  communication 34
  compilation nodes 5
  computation nodes 17
  synchronizing 5
  target 7, 9
  target nodes 4

O

Offline simulation 48
OpComm
  icon mask 38
OpComm Blocks
  OpSync_VME 32

OpComm blocks 6, 10
algebraic loop message. 17
for acquisition groups 17
in SC_ 17
in SM_ and SS_. 17
inserting 10, 15, 16, 17
OpAnalogIn 28
OpAnalogOut 29
OpDigitalIn 30
OpDigitalOut 31
OpQuad-Decoder 31
OpSync. 40
OpSync_VME. 32
OpTrigger 33
OpWriteFile 34
sampling times 42
Simulink parameters 17
SystemBuild parameters 19
OpWriteFile 34

P

Parameters
continuous modification 49
discrete modification 49
model performance 49
PC/104 cards. 27
PCI cards 27
Play button 55

Q

Quadrature decoders. 25
Quadrature meter 32

R

Real-time
calculations 13
communication 17
communication link 5, 17
continuous calculations 35
execution. 48
problems 48
simulation 6
target nodes 4
Real-Time Workshop 47
AutoCode 47
C code 47
ISI C code generator 47
makefile 47
rtdemo1 51
RT-LAB software
automatic code generation. 47
hardware configuration 4
interacting during simulations 7
using 6
RTW, see Real-Time Workshop
Running a simulation 48
Disconnect button. 56
multiprocessing system 7
Open Model button 53, 56
Play button. 55

S

Sampling rates
Delay 41
determining 41
transitions 41
Zero Order Hold 41
Sampling time
OpComms 42
Simulation
determining speed 11
executing. 55
interacting during 7, 49
modifying parameters 7
multiprocessing system 7
offline 25, 48
output. 31
running 48
speed 11
viewing. 7
Simulink 8
blocks in SC_ 13
blocks in SM_ 13
blocks in SS_ 13
library browser 26
multiple signals 34
multi-rate models 40
simulation speed 11
Single-Tasking mode 44
Single-Thread mode 40, 42
Slaves 9, 13
blocks. 13
Slider Gain 49
State Variables 10
Step Size
approximate 11
calculating 11
equation 11
increasing 34
minimum. 11

Subsystems 6
Console (SC_) 9, 13
defining 10, 13
Master (SM_) 9, 13
naming/organizing 14
prefixes 15
Slave (SS_) 9, 13
Synchronization 13, 33
algorithm 38
between nodes 5
Console time 38
hardware 12
I/O in multi-rate 45
I/O Synchronize option 27
modules 25
real-time 10
synchronized mode 34
synchronous acquisition 37
timers 12
SystemBuild 8
blocks in SC_ 13
blocks in SM_ 13
blocks in SS_ 13
palette 26
simulation speed 11
superblocks 15, 18
UCBs 19

T

Target nodes 4, 7
communication with Console 34
TCP/IP 7, 17
delay 49
Time Step, see Step Size
Timer card 12
Trigger 33

U

UCBs 19

V

Voltage 30

W

Windows NT 7

Z

Zero Order Hold 41, 45